Valgrind: Profiler and Program Checker

Alejandro Cabrera
Florida State University
Department of Computer Science
ac05k@fsu.edu

Overview

- Valgrind: What is it?
- Using Valgrind
- Valgrind Design and Implementation
- The Future of Valgrind

Definitions

- Profiler: A tool to investigate the behavior of the program as it runs.
- Checker: A subset of profilers. Used to verify certain program properties as they run.
- Static Analysis: Investigation of program source to determine certain properties about a program before it is ever run.
- Dynamic Analysis: Investigation of a program while it is running to determine certain properties.

Valgrind Trivia

- Pronounced Val-'grinned', not Val-'grind'.
- If Valgrind is not a shortening of the phrase “Value Grinder”, then where does the name come from?
- Nordic Mythology
  - Valgrind, the main entrance to Valhalla
  - Only those judged worthy are allowed entrance.

Valgrind, the *Mature* Profiler

- Currently at version 3.5.0, as of August 19, 2009
- Supports:
  - x86/Linux
  - AMD64/Linux
  - PPC32/Linux
  - PPC64/Linux
- In short, most platforms that you'll encounter.
- Support for ARM planned for near-future.

Valgrind, the *High-Profile* Profiler

- Not a toy: used in these major, production-level systems:
  - Mozilla Firefox
  - Sun OpenOffice
  - MySQL, PostgreSQL
  - GIMP
  - Python, Perl, PHP
  - OProfile (Linux Kernel profiler)
  - Boost C++ Libraries
  - Call of Duty
  - Much, much more...

Valgrind, the *Flexible* Profiler

- Includes several *tools* that enable various types of profiling:
  - memcheck – heap-allocated memory checking support
  - callgrind – function call profiling support
  - cachegrind – cache-behavior analysis
  - helgrind – threading behavior checker
  - massif – heap-profiler
  - ptrcheck – memory boundary checker for static, global, dynamic memory
  - Various others!
- All tools are plugins to the Valgrind core.
  - Adding a new tool is a matter of understanding the core.
- Supports execution of any language:
  - Merely need a supported binary executable to profile

Valgrind, the *Open* Profiler

- All source is publicly available.
- Design documents and various papers describe implementation and extension of Valgrind.

Using Valgrind

```
$> valgrind --tool=<tool_name> [tool_options] <exe_name>
```

- These slides will cover the usage of tools:
  - memcheck
  - callgrind
  - cachegrind

Memcheck: a memory error detector

- Useful for catching:
  - **Memory leaks**
  - Heap/stack overflow
  - Double-free
  - Overlapping memory regions in *cpy functions
  - Using undefined values

Using memcheck

```
$> valgrind --tool=memcheck [mc_options] <exe_name>
```

- Make sure your EXE is compiled without optimization, e.g. '-O#'.
- Adding '-g' to the compilation may result in more descriptive output from memcheck.

Understanding Memcheck's Output

- Illegal reads/writes
- Use of uninitialized values
- Use of invalid values in system calls
- Illegal frees
- Use of an incorrect deallocation function
- Overlapping source/destination
- Memory leaks

Understanding Memcheck's Output Illegal Reads/Writes

- Occurs as a result of your program performing a read or write in an illegal location.
  - Out of bounds, uninitialized memory, memory that does not belong to your program...
- Memcheck's output:

```
Invalid read of size 4
  at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40B07FF4: read_png_image(QImageIO *) (kernel/qpngio.cpp:326)
  by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

Understanding Memcheck's Output Unitialized Values

- Issued when your program tries to use undefined variables in such a way that affects *observable* behavior:
  - Output, branches, etc...
- Memcheck's output:

```
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)
```

Understanding Memcheck's Output Invalid Values in System Calls

- Checks all parameters passed to a system call.
- Memcheck's output:

```
Syscall param write(buf) points to uninitialised byte(s)
   at 0x25A48723: __write_nocancel (in /lib/tls/libc-2.3.3.so)
   by 0x259AFAD3: __libc_start_main (in /lib/tls/libc-2.3.3.so)
   by 0x8048348: (within /auto/homes/njn25/grind/head4/a.out)
 Address 0x25AB8028 is 0 bytes inside a block of size 10 alloc'd
   at 0x259852B0: malloc (vg_replace_malloc.c:130)
   by 0x80483F1: main (a.c:5)
Syscall param exit(error_code) contains uninitialised byte(s)
   at 0x25A21B44: __GI__exit (in /lib/tls/libc-2.3.3.so)
   by 0x8048426: main (a.c:8)
```

Understanding Memcheck's Output Invalid Frees

- Issued when a variable would be deallocated after a deallocation on it has already been performed.
- Memcheck's output:

```
Invalid free()
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
```

Understanding Memcheck's Output Incorrect Deallocation Functions

- Particularly important for C++
  - delete vs. delete[] for new vs. new[]
- Memcheck's output:

```
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)
```

Understanding Memcheck's Output Overlapping Source/Destination

- Indicates use of a set of overlapping memory addresses in a copying function.
  - strncpy(ptr, ptr+4, 8)
- Memcheck's output:

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

Understanding Memcheck's Output Memory Leaks

- Tracks all memory allocations and deallocations.
- Memcheck will know at the end of a program whether heap contains allocated memory.
- Four categories of memory leaks:
  - Still reachable: Memory that could have been free'd, but need not be.
  - Definitely lost: Memory leak. Fix it.
  - Indirectly lost: Set of pointers inaccessible, e.g., root node is free'd, children are not. Fix it.
  - Possibly lost: Could be a leak. Investigate carefully.

Understanding Memcheck's Output Memory Leaks

```
LEAK SUMMARY:
  definitely lost: 48 bytes in 3 blocks.
  indirectly lost: 32 bytes in 2 blocks.
    possibly lost: 96 bytes in 6 blocks.
  still reachable: 64 bytes in 4 blocks.
       suppressed: 0 bytes in 0 blocks.
```

- To get more detailed memcheck output for memory leaks, pass flags:
  - --leak-check=full --show-reachable=yes

Useful Memcheck Flags

- --leak-check=<no|summary|yes|full>
  - Default: summary
  - yes and full give details on individual leaks
- --show-reachable=<yes|no>
  - Default: no
  - By default, only *definitely lost* and *possibly lost* blocks are shown. If *yes*, adds *reachable* and *indirectly* lost to output.
- --track-origins=<yes|no>
  - Default: no
  - Indicates to memcheck to keep track of origin of all pointers. Output will inform you of exact source of memory errors.
  - Overhead: Requires at least 100MB more space and halves memcheck speed.

Memcheck Reliability

- Though no empirical evaluation has been conducted of the reliability of memcheck, developers claim very low false positive/negative rate.
- Likened to ignoring compiler warnings.

Callgrind: a Call-Graph Generating Cache Profiler

- Useful for determining bottleneck **functions** in a program
- Great for optimizing a program towards the end of development
- Also great for regression testing:
  - How do recent changes affect performance of program?

Using callgrind

```
$> valgrind --tool=callgrind [cg_options] <exe_name>
```

- EXE should be compiled with optimizations.
  - Performance testing!
- Adding '-g' to the compilation may result in more descriptive output from callgrind.
- Generates callgrind.out.<PID>, the program profile.

Example Callgrind Terminal Output

```
$> valgrind --tool=callgrind ./procinfo
...program output...
==15197==
==15197== Events    : Ir
==15197== Collected : 372711
==15197==
==15197== I   refs:      372,711
```

Resulting callgrind.out.15197: Header

```
$> cat ./callgrind.out.15197
...program output...
version: 1
creator: callgrind-3.5.0
pid: 15197
cmd:  ./procinfo
part: 1


...
```

Resulting callgrind.out.15197: Header 2

```
desc: I1 cache:
desc: D1 cache:
desc: L2 cache:

desc: Timerange: Basic block 0 - 75179
desc: Trigger: Program termination

positions: line
events: Ir
summary: 372711
```

Resulting callgrind.out.15197: A Function

```
fn=(3302) free
0 60
cfn=(2616)
calls=12 0
0 24
0 362
cfn=(3304) 0x00070c60
calls=10 0
0 838
0 50
```

Using callgrind_annotate to Make Sense of callgrind.out.15197

```
$> callgrind_annotate ./callgrind.out.15197

--------------------------------------------------------------------------------

Profile data file './callgrind.out.15197' (creator: callgrind-3.5.0)

--------------------------------------------------------------------------------

I1 cache:

D1 cache:

L2 cache:

Timerange: Basic block 0 - 75179

Trigger: Program termination

Profiled target:  ./procinfo (PID 15197, part 1)

...
```

Using callgrind_annotate to Make Sense of callgrind.out.15197

```
...
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  off
...
```

Using callgrind_annotate to Make Sense of callgrind.out.15197

```
--------------------------------------------------------------------------------
    Ir  file:function
--------------------------------------------------------------------------------
10,866  ???:strncmp [/lib/tls/i686/cmov/libc-2.9.so]
10,217  ???:vfprintf [/lib/tls/i686/cmov/libc-2.9.so]
 9,036  ???:memchr [/lib/tls/i686/cmov/libc-2.9.so]
 7,992  ???:fgets [/lib/tls/i686/cmov/libc-2.9.so]
 6,392  ???:_IO_getline_info [/lib/tls/i686/cmov/libc-2.9.so]
```

callgrind_annotate Flags

- callgrind_annotate [options] callgrind.out.<pid>
- Flags:
  - --auto=<yes|no> <default = yes>
    - Presents source code annotated with function call counts.
  - --tree=<none|caller|calling|both> <default = no>
    - Prints foe each function their calling or callee functions.
  - -I, --include=<DIR>
    - Adds DIR to search path for auto source code annotation.

Callgrind Usage Hints

- Avoiding cycles:
  - For recursive functions, callgrind_annotate may report incorrect costs/call counts.
  - Use KCachegrind to resolve.
- If used extensively and repeatedly, many profile outputs will result:
  - Add 'rm callgrind.out.*' to your build system **clean** section.

Cachegrind: A Cache and Branch-Prediction Profiler

- Allows for accurate profiling of total program instruction counts, branch misses, and L1/L2 cache misses.
- Profiles both the instruction cache and the data cache.
- Similar usage to callgrind tool.

Using cachegrind

```
$> valgrind --tool=cachegrind [cg_options] <exe_name>
```

- EXE should be compiled with optimizations.
  - Performance testing!
- Adding '-g' to the compilation may result in more descriptive output from callgrind.
- Generates cachegrind.out.<PID>, the program profile.

Example Cachegrind Terminal Output

```
$> valgrind --tool=cachegrind ./procinfo
...program output...
==15890== I   refs:      373,384
==15890== I1  misses:      1,934
==15890== L2i misses:      1,392
==15890== I1  miss rate:    0.51%
==15890== L2i miss rate:    0.37%
...
```

Example Cachegrind Terminal Output

```
…
==15890== D   refs:      195,202  (136,985 rd   + 58,217 wr)
==15890== D1  misses:      2,040  (  1,794 rd   +    246 wr)
==15890== L2d misses:      1,753  (  1,548 rd   +    205 wr)
==15890== D1  miss rate:     1.0% (    1.3%     +    0.4%  )
==15890== L2d miss rate:     0.8% (    1.1%     +    0.3%  )
==15890== L2 refs:         3,974  (  3,728 rd   +    246 wr)
==15890== L2 misses:       3,145  (  2,940 rd   +    205 wr)
==15890== L2 miss rate:      0.5% (    0.5%     +    0.3%  )
```

Resulting cachegrind.out.15890

```
$> cat cachegrind.out.15890
desc: I1 cache:         65536 B, 64 B, 2-way associative
desc: D1 cache:          65536 B, 64 B, 2-way associative
desc: L2 cache:          262144 B, 64 B, 8-way associative
cmd: ./procinfo
events: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
...
```

Resulting cachegrind.out.15890

```
...
fn=main
496 4 1 1 0 0 0 1 0 0
500 6 0 0 2 0 0 4 0 0
502 2 0 0 1 0 0 0 0 0
503 3 0 0 0 0 0 2 0 0
504 3 1 1 0 0 0 2 0 0
505 4 0 0 0 0 0 2 0 0
522 1 1 1 0 0 0 0 0 0
523 2 0 0 2 0 0 0 0 0
...
```

Using cg_annotate to Make Sense of cachegrind.out.15890

```
$> cg_annotate ./cachegrind.out.15890
--------------------------------------------------------------------------------
I1 cache:         65536 B, 64 B, 2-way associative
D1 cache:          65536 B, 64 B, 2-way associative
L2 cache:         262144 B, 64 B, 8-way associative
Command:           ./procinfo
Data file:        ./cachegrind.out.15890
Events recorded:  Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Events shown:     Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Event sort order: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Thresholds:       99 0 0 0 0 0 0 0 0
Include dirs:
User annotated:
Auto-annotation:  off
...
```

Using cg_annotate to Make Sense of cachegrind.out.15890

```
…
--------------------------------------------------------------------------------
    Ir  I1mr  I2mr      Dr  D1mr  D2mr     Dw D1mw D2mw
--------------------------------------------------------------------------------
373,384 1,934 1,392 136,985 1,794 1,548 58,217  246  205  PROGRAM TOTALS


--------------------------------------------------------------------------------
    Ir  I1mr I2mr     Dr  D1mr D2mr     Dw D1mw D2mw  file:function
--------------------------------------------------------------------------------
188,555 1,005  836 68,916 1,051  860 26,704  201  179  ???:???
 75,409    14   14 31,748   633  603 10,267    2    0  ???:_dl_addr
 13,394    12   12  5,326     1    1  1,636    0    0  ???:_IO_file_xsputn
 13,392   186  110  4,919    21   15  2,245   11    0  ???:_IO_vfscanf
 10,866   125    5  3,741     1    1  1,566    0    0  ???:strncmp
…
```

cg_annotate Flags

- cg_annotate [options] cachegrind.out.<pid>
- Flags:
  - --auto=<yes|no> <default = yes>
    - Presents source code annotated with function call counts.
  - -I, --include=<DIR>
    - Adds DIR to search path for auto source code annotation.
  - --show=<A,B,C>
  - --sort=<A,B,C>
    - Allows customization of output.
    - List can include any of: Ir, I1mr, I2mr, D1mr, D2mr, …

Effective Profiling with Cachegrind

- Output:
  - Sort by D2mr: often times, the misses to the L2 data cache dominate program execution time.
- Run with cache cold, then with cache hot:
  - Useful for determining how to optimize start up (cold).
  - Useful for optimizing typical run (hot).
- Optimizing branching:
  - Observe Bim and Bcm, particularly in presence of many *switch* statements.
  - May be suggestive of using table-driven decision-making.

Cachegrind Limitations

- Doesn't account for kernel activity.
- Alters thread scheduling for multi-threaded programs.
- Sensitive to address space randomization.
- Sensitive to executable size.
- Sensitive to multiple runs.

Valgrind Design and Implementation

- More Definitions
- Shadow Values
- Valgrind Core

Definitions

- Dynamic Binary Analysis (DBA)– Analysis of behavior of a running binary-compiled executable.
- Dynamic Binary Instrumentation (DBI) – An implementation of DBA. Analytical code is inserted into user code at run-time.
- Shadow Values – Software simulated registers containing information about physical register/memory values. One-to-one mapping at bit, byte, or word granularity.

Overhead of Profiling

- May increase program execution time by up to 10-100x original run-time.
- Why would developers pay so much to profile?
  - High probability that profiler will catch errors that are nearly impossible to catch otherwise
  - Profiler optimizes process of program optimization by highlighting primary bottlenecks

Shadow Values and DBI

- Allows for heavyweight DBI
- Difficult to implement correctly
  - Nine requirements are identified by Valgrind developers

Shadow Value Requirements

R1. Provide shadow registers

R2. Provide shadow memory

R3. Instrument read/write instructions

R4. Instrument read/write system calls

R5. Instrument startup allocations

R6. Instrument system call (de)allocations

R7. Instrument stack (de)allocations

R8. Instrument heap (de)allocations

R9. Provide channels for profiler-generated output

Shadow Values and Valgrind

- Full implementation of shadow values allows Valgrind to perform virtually any type of DBA.
- Making the state of memory, registers, and the ability to instrument most read/write and memory operations grants extensive information to tool developers.

Valgrind Shadow Value Support

- R1: Shadow registers are first-class entities
  - Unlimited temporaries, exposed intermediate values, easy to operate on
- R2: Not explicitly supported, but facilitated by thread synchronization
- R3: All reads/writes instrument-able
  - Aided by Valgrind's IR disassembly
- R4-R7: Supported by Valgrind's event system
- R8: Tools using Valgrind core must use wrappers to alloc/dealloc funcs. to instrument
- R9: Allows tools to specify output channel

Code Representation for Profilers

- Two methods:
  - D & R (Disassemble-and-Resynthesize): IR-based approach, much akin to compilers.
  - C & A (Copy-and-Annotate): Copies machine instructions verbatim, inserting annotations to perform instrumentation.
- Valgrind uses D&R – more powerful, yet more difficult to implement.

Valgrind Instrumentation Phases

- Multiple phases used to convert machine code to Valgrind IR:
  - Phase 1: Disassembly: machine code → tree IR
  - Phase 2: Optimize 1: tree IR → flat IR
  - Phase 3: Instrumentation: flat IR → flat IR
  - Phase 4: Optimize 2: flat IR → flat IR
  - Phase 5: Tree-build: flat IR → tree IR
  - Phase 6: Instruction select: tree IR → Instr. List
  - Phase 7: Register Allocation: Instr. List → Instr. List
  - Phase 8: Assembly: Instr. List → Machine Code
- Essentially, a JIT-compiler

Valgrind Implementation Detail: Event System

- Used to handle case of system calls, since Valgrind does not trace into kernel
- Provides tool developers with a set of pre_* ad post_* functions that given a system call to watch for, executes a callback function before and after that system call
- Valgrind provides wrappers for all system calls
  - “15,000 lines of tedious C code...”
- Event system also important for stack (de)allocations

Valgrind Implementation Detail: Handling Threaded Code

- Threaded programs introduce race-conditions between shadow values and actual values:
  - A threaded load-store on the actual code may not correspond to the same load-store on the shadow values.
  - Need to introduce synchronization
- Currently, Valgrind synchronizes all accesses to shadow values, effectively serializing threaded programs.
- **Open problem: Very slow.**
  - How to implement shadow value based DBA framework that behaves correctly in presence of threads and scales well?

Valgrind Robustness

- Robustness – A quality indicating correct output in all scenarios
- How robust is Valgrind?
  - How do you quantify *robustness*?
  - False positives + false negative counts?
  - Comparison to related works?
- Currently, no empirical study of Valgrind's robustness has been conducted
  - Only anecdotal evidence and theoretical claims exist

Valgrind Robustness Qualities in Favor

- Widely used:
  - Therefore, widely tested
- Independent of libc:
  - All standard functions implemented within Valgrind using system calls
- Thread-safe
- Trade off: speed for accuracy
  - Program run using memcheck tool runs on average 22.2x slower than normal.
  - Shadow memory operations are slow

Research: Open Questions

- How to implement scalable, parallel/concurrent shadow value manipulation?
- More compiler optimizations at D&R phase
- Empirical evaluation of *robustness* of Valgrind and associated tools
- Performance improvements on Valgrind to reduce execution slowdown

Future Development Possibilities

- Additional tools to extend use of Valgrind's unique heavyweight tool support
- Support for more languages in auto_annotation:
  - Currently supports C, C++, assembly, and Fortran
- Support more platforms:
  - ARM is becoming an extremely prominent platform with the advent of smart, mobile computing
- Support for more operating systems:
  - Windows? Mac OS?

A Quote: Nicholas Nethercote

"Ever since programming began, programming tools have slowly improved, making the difficult job of writing good programs easier. Dynamic binary analysis tools such as Memcheck, Cachegrind, Annelid, and Redux, built with dynamic binary instrumentation frameworks such as Valgrind, are just another step in this progression. I hope that they will help many programmers improve their programs. But I look forward more to the day when the art of programming has progressed such that they are no longer necessary, having been eclipsed by something even better."

References

- The Valgrind Developers. Valgrind. http://www.valgrind.org/
- Nicholas Nethercote and Julian Seward. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. PLDI' 07.