Valgrind: A Program Supervision Framework

Aaron Evans
2004 Nov 15
Citation


http://www.sciencedirect.com/science/article/B75H1-4DDWKTJ-PG/2/49e9f28ff4e74ceeb8e34e4bf4050f5b
How do you pronounce “Valgrind”? 

The “Val” as in the world “value”. The “grind” is pronounced with a short ‘i’ -- ie. “grinned” (rhymes with “tinned”) rather than “grined” (rhymes with “find”).

Don’t feel bad: almost everyone gets it wrong at first.

http://valgrind.kde.org/faq.html
Where does the name “Valgrind” come from?

Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance.

It’s not short for “value grinder”, although that’s not a bad guess.

http://valgrind.kde.org/faq.html
Valgrind History

- First released in 2002
- It was originally a memory checker
- Became a “meta-tool”
Valgrind Overview

• a meta-tool enabling program supervision
• core - performs binary translation of x86 instructions
• skins - interface to core to check execution
Agenda

1. Introduction
2. Valgrind Core
3. Valgrind Skins
4. A Valgrind “skin” - Memcheck
5. Performance
6. Conclusions
Valgrind Core

• Valgrind works with ordinary dynamically-linked executables (client)
• Core dynamically translates x86 to UCode to x86
• UCode is RISC-like, two-address immediate language
• Checkers check UCode
UCode

• UCode uses a simulated register set
• valgrind holds state for the virtual processor
  - simulated registers
  - condition codes for registers
• simulated state is updated at the end of each basic block
Translating Basic Blocks

1. disassemble x86 to UCode
2. optimize UCode
3. instrument UCode
4. allocate registers
5. translate to x86
6. execute instrumented x86 code
### Translation Example
(disassembly: x86 → UCode)

<table>
<thead>
<tr>
<th>Assembly</th>
<th>UCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xFFF, %ebx</td>
<td>0: MOVL $0xFFF, t0</td>
</tr>
<tr>
<td>andl %ebx, %eax</td>
<td>1: PUTL t0, %EBX</td>
</tr>
<tr>
<td>ret</td>
<td>2: INCEIPo $5</td>
</tr>
</tbody>
</table>

---

1: PUTL t0, %EBX
2: INCEIPo $5
3: GETL %EAX, t2
4: GETL %EBX, t4
5: ANDL t4, t2 (-wOSZACP)
6: PUTL t2, %EAX
7: INCEIPo $2
8: GETL %ESP, t6
9: LDL (t6), t8
10: ADDL $0x4, t6
11: PUTL t6, %ESP
12: JMPo-r t8
## Translation Example (optimization)

<table>
<thead>
<tr>
<th>movl $0xFFF</th>
<th>0: MOVL $0xFFFF, t0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1: PUTL t0, %EBX</td>
</tr>
<tr>
<td></td>
<td>2: INCEIPO $5</td>
</tr>
<tr>
<td>andl %ebx, %eax</td>
<td>3: GETL %EAX, t2</td>
</tr>
<tr>
<td></td>
<td>4: GETL %EBX, t4</td>
</tr>
<tr>
<td></td>
<td>5: ANDL t4, t2 (-wOSZACP)</td>
</tr>
<tr>
<td></td>
<td>6: PUTL t2, %EAX</td>
</tr>
<tr>
<td></td>
<td>7: INCEIPO $2</td>
</tr>
<tr>
<td>ret</td>
<td>8: GETL %ESP, t6</td>
</tr>
<tr>
<td></td>
<td>9: LDL (t6), t8</td>
</tr>
<tr>
<td></td>
<td>10: ADDL $0x4, t6</td>
</tr>
<tr>
<td></td>
<td>11: PUTL t6, %ESP</td>
</tr>
<tr>
<td></td>
<td>12: JMPo-r t8</td>
</tr>
</tbody>
</table>
### Translation Example

(optimization)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0:</th>
<th>MOVL</th>
<th>$0xFFFF, t0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1:</td>
<td>PUTL</td>
<td>t0, %EBX</td>
</tr>
<tr>
<td></td>
<td>2:</td>
<td>INCEIPO</td>
<td>$5</td>
</tr>
<tr>
<td>andl %ebx, %eax</td>
<td>3:</td>
<td>GETL</td>
<td>%EAX, t2</td>
</tr>
<tr>
<td></td>
<td>4:</td>
<td>GETL</td>
<td>%EBX, t4</td>
</tr>
<tr>
<td></td>
<td>5:</td>
<td>ANDL</td>
<td>t0, t2 (-wOSZACP)</td>
</tr>
<tr>
<td></td>
<td>6:</td>
<td>PUTL</td>
<td>t2, %EAX</td>
</tr>
<tr>
<td>ret</td>
<td>7:</td>
<td>INCEIPO</td>
<td>$2</td>
</tr>
<tr>
<td></td>
<td>8:</td>
<td>GETL</td>
<td>%ESP, t6</td>
</tr>
<tr>
<td></td>
<td>9:</td>
<td>LDL</td>
<td>(t6), t8</td>
</tr>
<tr>
<td></td>
<td>10:</td>
<td>ADDL</td>
<td>$0x4, t6</td>
</tr>
<tr>
<td></td>
<td>11:</td>
<td>PUTL</td>
<td>t6, %ESP</td>
</tr>
<tr>
<td></td>
<td>12:</td>
<td>JMPo-r</td>
<td>t8</td>
</tr>
</tbody>
</table>
### Translation Example (instrumentation)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0xFFF</td>
<td>0</td>
<td>MOVL $0xFFFF, t0</td>
</tr>
<tr>
<td>putl t0, %EBX</td>
<td>1</td>
<td>PUTL t0, %EBX</td>
</tr>
<tr>
<td>inceipo $5</td>
<td>2</td>
<td>INCEIPO $5</td>
</tr>
<tr>
<td>andl %ebx, %eax</td>
<td>3</td>
<td>GETL %EAX, t2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>GETL %EBX, t4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>ANDL t0, t2 (-wOSZACP)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>PUTL t2, %EAX</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>INCEIPO $2</td>
</tr>
<tr>
<td>ret</td>
<td>8</td>
<td>GETL %ESP, t6</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>LDL (t6), t8</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>ADDL $0x4, t6</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>PUTL t6, %ESP</td>
</tr>
<tr>
<td>jmpo-r t8</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Translation Example
(register allocation)

movl $0xFFF
andl %ebx, %eax
ret

| 0: MOVL | $0xFFFF, %eax |
| 1: PUTL | %eax, %EBX |
| 2: INCEIPO | $5 |
| 3: GETL | %EAX, %ebx |
| 4: ANDL | %eax, %ebx(-wOSZACP) |
| 5: PUTL | %ebx, %EAX |
| 7: INCEIPO | $2 |
| 8: GETL | %ESP, %ecx |
| 9: LDL | (%ecx), %edx |
| 10: ADDL | $0x4, %ecx |
| 11: PUTL | %ecx, %ESP |
| 12: JMPo-r | %edx |
Translation Example
(code generation: UCode $\rightarrow$ x86)

0: MOVL $0xFFFF, %eax
1: PUTL %eax, %EBX
2: INCEIPO $5
3: GETL %EAX, %ebx
4: ANDL %eax,%ebx (−wOSZACP)
5: PUTL %ebx, %EAX
6: INCEIPO $2
7: GETL %ESP, %ecx
8: LDL (%ecx), %edx
9: ADDL $0x4, %ecx
10: PUTL %ecx, %ESP
11: JMPo-r %edx

movl $0xFFFF, %eax
movl %eax, 0xC(%ebp)
movb $0x18, 0x24(%ebp)
movl $0x0(%ebp), %ebx
andl %eax, %ebx
movl %ebx, 0x0(%ebp)
movb $0x1A, 0x24(%ebp)
movl 0x10(%ebp), %ecx
movl (%ecx), %edx
pushfl; popl 32(%ebp)
addl $0x4, %ecx
movl %ecx, 0x10(%ebp)
movl %edx, %eax
ret
Connecting Basic Blocks

• Translated basic blocks are cached
• Cache holds ~160,000 basic blocks
• At the end of a basic block,
  - jumps to address known at compile-time (chain, 70%)
  - address not known at compile time
    • translated block in cache
    • untranslated block
System Calls

- System calls are not converted to UCode
- The core does the following for a syscall:
  i. save valgrind’s stack pointer
  ii. copy simulated registers (except PC) into real registers
  iii. do the system call
  iv. copy simulated registers out to memory (except PC)
  v. restore valgrind’s stack pointer
Floating Point, MMX, SSE, etc

- load simulated FPU state into the FPU
- execute
- copy FPU state to the simulated state
- similar approach for MMX, SSE, etc
Client-requests

• a “trapdoor” for clients to query core
• client code contains trapdoor instruction sequence
• core identifies sequence and waits for client request via signal
Ensuring Correctness

- in x86→UCode→x86’, is x86 functionally equivalent to x86’?
- no formal way to prove correctness
- valgrind can revert to CPU execution to pinpoint problems
Signals

- valgrind should receive signals that are sent to clients
- valgrind intercepts a client's `sigaction()` and `sigprocmask()` and registers the signals for itself
- periodically, valgrind delivers any pending signals
- “deliver”
  - build stack frame at intended client code
  - execute,
  - upon return, continue from prior location
Threads

• How should threads be modeled?
  - one valgrind thread per client thread?
  - complex due to 1) thread-safety between valgrind structures 2) thread-safety between skins and core
  - consider memcheck

• Solution: only support pthreads, use custom pthread lib
  - valgrind controls context switching within a single thread
  - reimplementation of libpthread complicates the core
Skins

- Skins define instrumentation of UCode
- A client program has three levels of control:
  - user space: all JIT compiled code
  - core space: signal handling, pthreads, scheduling
  - kernel space: execution in kernel
Programming Skins

• Each skin is a shared object
• A programmer of a skin must define four functions:
  - initialization (2)
  - instrumentation
  - finalization
Initialization
Functions

• Details: name, copyright, etc
• Needs: list of services needed from core
• Trackable Events: indicate which core events are of interested to the skin
Instrumentation

• upon translation, function is called to instrument UCode
• typically, instrumentation is just a function call
• it’s possible to define new UCode instructions
Finalization

• a finalization function is called per skin to output results
Overriding Library Functions

• skins can override library functions
An Example: Memcheck

memcheck can detect:
- use of uninitialized memory
- accessing memory after it has been freed
- accessing memory past the end of heap blocks
- accessing inappropriate areas on the stack
- Memory leaks- pointers to heap blocks are lost
- passing of uninitialized /unaddressable memory to syscalls
- mismatched malloc() / new / new[] vs. free() / delete / delete[]
- overlapping source and destination areas for memcpy(), strcpy(), etc
Memcheck Overview

• each byte of memory is shadowed with nine status bits
• ‘A’ bit - whether or not a byte is addressable
• 8 ‘V’ bits - which bytes have defined values (based on C semantics)
  - allows bit-field operations to be accurately checked
Services Used

• error recording - skin provides functions for reporting errors
• debug information - core provides functions that take an address and return debug info
• shadow registers - skin defines one function that defines the valid bits for shadow register
• client requests - if the core receives an unrecognizable client request, it is passed to the skins
• extended UCode - inlines instrumentation
• replacement library functions - memcheck replaces malloc, free, etc
Events Tracked

- `mma()`, `brk()`, `mprotect()`, `mremap()`, `munmap()`
- A and v bits are checked before system calls that read memory
- v bits are updated after all those that write memory
Instrumentation

• For every UCode instruction, instrumented code is added immediately before it.

• Most instrumentation updates or checks for consistency of A and V bits for memory and registers.
Performance

- Tested on 1400MHz Athlon, 1GB RAM
- testing (some) SPEC2000 benchmarks
## Performance

(slowdown)

<table>
<thead>
<tr>
<th>Program</th>
<th>Time (s)</th>
<th>Nulgrind</th>
<th>Memcheck</th>
<th>Addrcheck</th>
<th>Cacheegrind</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>10.7</td>
<td>2.4</td>
<td>13.6</td>
<td>9.1</td>
<td>31.0</td>
</tr>
<tr>
<td>crafty</td>
<td>3.5</td>
<td>7.2</td>
<td>44.6</td>
<td>26.5</td>
<td>107.4</td>
</tr>
<tr>
<td>gap</td>
<td>0.9</td>
<td>5.4</td>
<td>28.7</td>
<td>14.4</td>
<td>46.6</td>
</tr>
<tr>
<td>gcc</td>
<td>1.5</td>
<td>8.5</td>
<td>36.2</td>
<td>23.6</td>
<td>73.2</td>
</tr>
<tr>
<td>gzip</td>
<td>1.8</td>
<td>4.4</td>
<td>20.8</td>
<td>14.5</td>
<td>50.3</td>
</tr>
<tr>
<td>mcf</td>
<td>0.3</td>
<td>2.1</td>
<td>11.6</td>
<td>5.9</td>
<td>18.5</td>
</tr>
<tr>
<td>parser</td>
<td>3.3</td>
<td>3.7</td>
<td>17.4</td>
<td>12.5</td>
<td>34.8</td>
</tr>
<tr>
<td>twolf</td>
<td>0.2</td>
<td>5.2</td>
<td>29.2</td>
<td>18.5</td>
<td>53.3</td>
</tr>
<tr>
<td>vortex</td>
<td>6.5</td>
<td>7.5</td>
<td>47.9</td>
<td>32.7</td>
<td>88.4</td>
</tr>
<tr>
<td>ammp</td>
<td>18.9</td>
<td>1.8</td>
<td>24.8</td>
<td>21.1</td>
<td>47.1</td>
</tr>
<tr>
<td>art</td>
<td>26.1</td>
<td>5.9</td>
<td>14.1</td>
<td>11.5</td>
<td>19.4</td>
</tr>
<tr>
<td>equake</td>
<td>2.1</td>
<td>5.5</td>
<td>32.7</td>
<td>28.0</td>
<td>49.9</td>
</tr>
<tr>
<td>mesa</td>
<td>2.7</td>
<td>4.7</td>
<td>41.9</td>
<td>31.6</td>
<td>64.5</td>
</tr>
<tr>
<td>median</td>
<td>5.2</td>
<td>28.7</td>
<td>18.5</td>
<td>9.98</td>
<td>9.98</td>
</tr>
</tbody>
</table>
# Performance

*(code expansion)*

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (KB)</th>
<th>Nulgrind</th>
<th>Memcheck</th>
<th>Addrcheck</th>
<th>Cache_check</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>34</td>
<td>5.2</td>
<td>12.1</td>
<td>6.8</td>
<td>9.1</td>
</tr>
<tr>
<td>crafty</td>
<td>156</td>
<td>4.5</td>
<td>10.9</td>
<td>5.9</td>
<td>8.2</td>
</tr>
<tr>
<td>gap</td>
<td>140</td>
<td>5.6</td>
<td>12.7</td>
<td>7.3</td>
<td>9.7</td>
</tr>
<tr>
<td>gcc</td>
<td>564</td>
<td>5.9</td>
<td>13.1</td>
<td>7.6</td>
<td>9.9</td>
</tr>
<tr>
<td>gzip</td>
<td>30</td>
<td>5.5</td>
<td>12.6</td>
<td>7.2</td>
<td>9.4</td>
</tr>
<tr>
<td>mcf</td>
<td>30</td>
<td>5.7</td>
<td>13.5</td>
<td>7.7</td>
<td>9.9</td>
</tr>
<tr>
<td>parser</td>
<td>97</td>
<td>6.0</td>
<td>13.6</td>
<td>7.8</td>
<td>10.1</td>
</tr>
<tr>
<td>twolf</td>
<td>114</td>
<td>5.2</td>
<td>12.2</td>
<td>7.0</td>
<td>9.3</td>
</tr>
<tr>
<td>vortex</td>
<td>234</td>
<td>5.8</td>
<td>13.2</td>
<td>8.1</td>
<td>10.1</td>
</tr>
<tr>
<td>ammp</td>
<td>68</td>
<td>4.7</td>
<td>11.7</td>
<td>7.1</td>
<td>9.5</td>
</tr>
<tr>
<td>art</td>
<td>24</td>
<td>5.5</td>
<td>13.0</td>
<td>7.5</td>
<td>9.8</td>
</tr>
<tr>
<td>equake</td>
<td>44</td>
<td>5.0</td>
<td>12.2</td>
<td>7.1</td>
<td>9.2</td>
</tr>
<tr>
<td>mesa</td>
<td>69</td>
<td>4.8</td>
<td>11.2</td>
<td>6.7</td>
<td>8.9</td>
</tr>
<tr>
<td>median</td>
<td></td>
<td>5.5</td>
<td>12.6</td>
<td>7.2</td>
<td>9.5</td>
</tr>
</tbody>
</table>
Tools built with Valgrind

• KCacheGrind- collect call tree information
Tools built with Valgrind

- VGprof - profiler
- Redux - creates dynamic dataflow graphs
Conclusions

valgrind...

• works with compiled programs
• dynamically compiles x86 to UCode
• provides a skin interface for arbitrary instrumentation of UCode
• has acceptable performance
• has been used for a variety of purposes