Valgrind: A Program Supervision Framework

Aaron Evans

2004 Nov 15

Citation

Nicholas Nethercote and Julian Seward, Valgrind: A Program Supervision Framework, Electronic Notes in Theoretical Computer Science, Volume 89, Issue 2, October 2003, Pages 1-23.

`http://www.sciencedirect.com/science/article/B75H1-4DDWKTJ-PG/2/49e9f28ff4e74ceeb8e34e4bf4050f5b`

Preliminaries (1/2)

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

Preliminaries (2/2)

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)

`movl $0xFFF, %ebx`	`0: MOVL`	`$0xFFFF, t0`
	`1: PUTL`	`t0, %EBX`
	`2: INCEIPo`	`$5`
`andl %ebx, %eax`	`3: GETL`	`%EAX, t2`
	`4: GETL`	`%EBX, t4`
	`5: ANDL`	`t4, t2 (-wOSZACP)`
	`6: PUTL`	`t2, %EAX`
	`7: INCEIPo`	`$2`
`ret`	`8: GETL`	`%ESP, t6`
	`9: LDL`	`(t6), t8`
	`10: ADDL`	`$0x4, t6`
	`11: PUTL`	`t6, %ESP`
	`12: JMPo-r`	`t8`

Translation Example

(optimization)

movl $0xFFF	0: MOVL	$0xFFFF, t0
	1: PUTL	t0, %EBX
	2: INCEIPo	$5
andl %ebx, %eax	3: GETL	%EAX, t2
	4: ~~GETL~~	~~%EBX, t4~~
	5: ANDL	t4, t2 (-wOSZACP)
	6: PUTL	t2, %EAX
	7: INCEIPo	$2
ret	8: GETL	%ESP, t6
	9: LDL	(t6), t8
	10: ADDL	$0x4, t6
	11: PUTL	t6, %ESP
	12: JMPo-r	t8

Translation Example

(optimization)

`movl $0xFFF`	`0: MOVL`	`$0xFFFF, t0`
	`1: PUTL`	`t0, %EBX`
	`2: INCEIPo`	`$5`
`andl %ebx, %eax`	`3: GETL`	`%EAX, t2`
	4: ~~`GETL`~~	~~`%EBX, t4`~~
	`5: ANDL`	`t0, t2 (-wOSZACP)`
	`6: PUTL`	`t2, %EAX`
	`7: INCEIPo`	`$2`
`ret`	`8: GETL`	`%ESP, t6`
	`9: LDL`	`(t6), t8`
	`10: ADDL`	`$0x4, t6`
	`11: PUTL`	`t6, %ESP`
	`12: JMPo-r`	`t8`

Translation Example

(instrumentation)

movl $0xFFF	0: MOVL $0xFFFF, t0
	1: PUTL t0, %EBX
	2: INCEIPo $5
andl %ebx, %eax	3: GETL %EAX, t2
	4: GETL %EBX, t4
	5: ANDL t0, t2 (-wOSZACP)
	6: PUTL t2, %EAX
	7: INCEIPo $2
ret	8: GETL %ESP, t6
	9: LDL (t6), t8
	10: ADDL $0x4, t6
	11: PUTL t6, %ESP
	12: JMPo-r t8

Translation Example
(register allocation)

x86	UCode	
movl $0xFFF	0: MOVL	$0xFFFF, %eax
	1: PUTL	%eax, %EBX
	2: INCEIPo	$5
andl %ebx, %eax	3: GETL	%EAX, %ebx
	4: ANDL	%eax, %ebx(-wOSZACP)
	5: PUTL	%ebx, %EAX
	7: INCEIPo	$2
ret	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 → x86)

0: MOVL	$0xFFFF, %eax	movl	$0xFFFF, %eax
1: PUTL	%eax, %EBX	movl	%eax, 0xC(%ebp)
2: INCEIPo	$5	movb	$0x18, 0x24(%ebp)
3: GETL	%EAX, %ebx	movl	$0x0(%ebp), %ebx
4: ANDL	%eax,%ebx (-wOSZACP)	andl	%eax, %ebx
5: PUTL	%ebx, %EAX	movl	%ebx, 0x0(%ebp)
6: INCEIPo	$2	movb	$0x1A, 0x24(%ebp)
7: GETL	%ESP, %ecx	movl	0x10(%eb), %ecx
8: LDL	(%ecx), %edx	movl	(%ecx), %edx
9: ADDL	$0x4, %ecx	pushfl;	popl 32(%ebp)
		addl	$0x4, %ecx
10: PUTL	%ecx, %ESP	movl	%ecx, 0x10(%ebp)
11: JMPo-r	%edx	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:
  1. save valgrind's stack pointer
  2. copy simulated registers (except PC) into real registers
  3. do the system call
  4. copy simulated registers out to memory (except PC)
  5. 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 clients `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)

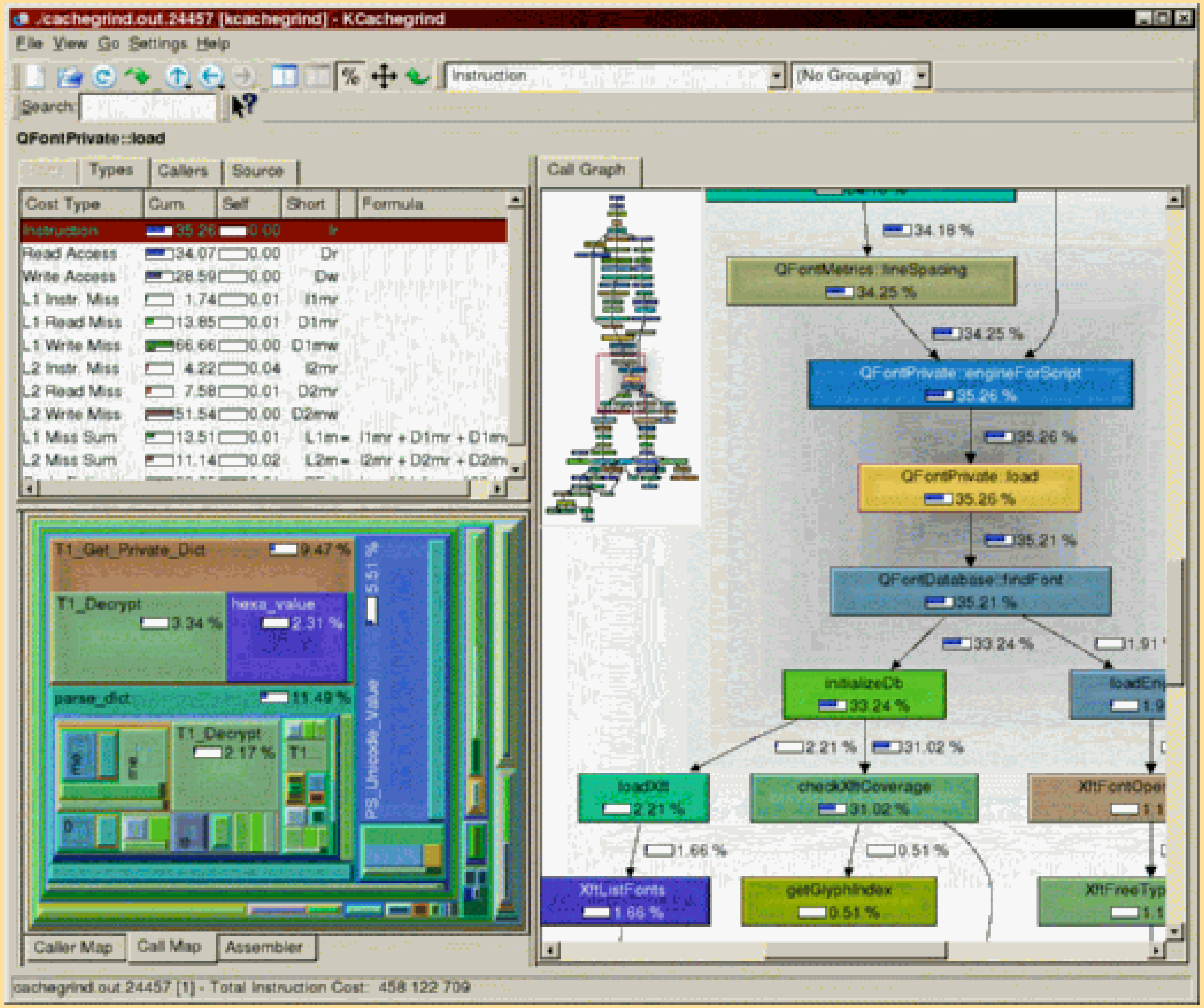
Program	Time (s)	Nulgrind	Memcheck	Addrcheck	Cachegrind
bzip2	10.7	2.4	13.6	9.1	31.0
crafty	3.5	7.2	44.6	26.5	107.4
gap	0.9	5.4	28.7	14.4	46.6
gcc	1.5	8.5	36.2	23.6	73.2
gzip	1.8	4.4	20.8	14.5	50.3
mcf	0.3	2.1	11.6	5.9	18.5
parser	3.3	3.7	17.4	12.5	34.8
twolf	0.2	5.2	29.2	18.5	53.3
vortex	6.5	7.5	47.9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

Performance
(code expansion)

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

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