Can We Make Compilers That Work?

John Regehr
September 2010
• **Undergrad**
  – Kansas State 1990-1995
  – Math and computer science

• **Grad school**
  – University of Virginia 1995-2001
  – 1 summer internship at a small company
  – 2 summer internships at Microsoft Research

• **Postdoctoral researcher**
  – Utah CS 2001-2003

• **On the faculty at Utah CS since 2003**
• Reported 277 bugs to teams developing C compilers
  – Most have been fixed

• Found serious wrong-code bugs in all C compilers we’ve tested
  – Including those used to compile safety-critical embedded systems
  – Including 6 bugs in a compiler that was proved to be correct
• What’s going on here?
  – Why can’t anyone create a C compiler that we can’t break?
• Our goal: Robust open-source compilation tools
  – We keep finding and reporting bugs until we stop finding them
  – Hasn’t happened after 2.5 years...

• What about commercial compilers?
static int x;
static int *volatile z = &x;
static int foo (int *y) {
    return *y;
}

int main (void) {
    *z = 1;
    printf ("%d\n", foo(&x));
    return 0;
}

• Should print “1”
• GCC rev 164319 at –O2 on x86-64 prints “0”
• Do compiler bugs even matter?
  – Students in my embedded systems courses routinely encounter compiler bugs
  – Large development efforts routinely encounter compiler bugs
  – C compiler is part of the trusted computing base for most computer systems
• **Symptoms of compiler bugs**
  1. Failure to emit code
  2. Emitted code crashes or computes wrong result
  3. Emitted code violates the *volatile invariant*

• **All tested compilers have bugs with all three kinds of symptoms**
Test case generator

C program

Compiler 1

Compiler 2

Compiler 3

...
Test Case Generator

• Grammar for C subset

• Lots of constraints
  – Must declare a variable before using it
  – Etc.

• Generator is driven by...
  – Random search
  – Depth first search
int foo (int x) {  
    return (x+1) > x;
}

int main (void) {  
    printf ("%d\n",       
        foo (INT_MAX));
    return 0;
}
# Not a Bug #2

```c
int bar (int x) {
    int i;
    if (i > 10) x++;
    return x;
}

int main (void) {
    printf("%d\n", bar (50));
}
```

```
$ clang -O0 init.c -o init
$ ./init
51
$ clang -O1 init.c -o init
$ ./init
50
```
Not a Bug #3

```
#include <stdio.h>

int main (void) {
    long a = -1;
    unsigned b = 1;
    printf("%d\n", a > b);
    return 0;
}
```

```
$ gcc compare.c -o compare
$ ./compare
0
$ gcc -m32 compare.c -o \
    compare
$ ./compare
1
```
• Property we require:
  – Anytime changing the compiler or optimization level changes the program’s result, it’s a compiler bug

• Without this property, automated testing is impossible

• Generated code must not...
  – Execute undefined behavior (191 kinds)
  – Rely on unspecified behavior (52 kinds)
Less undefined / unspecified behavior

Lindig 07

Our work

McKeeman 98

Less expressive

More expressive

Sheridan 07

More undefined / unspecified behavior
Supported features:

- Arithmetic, logical, and bit operations on integers
- For loops
- Conditionals
- Function calls
- Const and volatile
- Structs
- Pointers and arrays
- Goto
- Switch
- Break, continue
- Bitfields

Can easily add:

- Side-effecting expressions
- Comma operator

Probably not anytime soon:

- Interesting type casts
- Strings
- Unions
- Floating point
- Nontrivial C++
- Nonlocal jumps
- Varargs
- Recursive functions
- Function pointers
- Dynamic memory alloc.
Avoiding Undefined and Unspecified Behavior

• Offline avoidance is too difficult
  – E.g. ensuring in-bounds array access

• Online avoidance is too inefficient
  – E.g. ensuring validity of pointer to stack

• Solution: Combine static analysis and dynamic checks
Order of Evaluation Problems

• Order of evaluation of function arguments is unspecified

• E.g.

  \texttt{foo(bar(),baz())}

• Where \texttt{bar()} and \texttt{baz()} both modify some variable
Order of Evaluation Problems

• Solution:
  – Interprocedural analysis to compute conservative read and write set for each function
  – In between sequence points, never invoke functions where read and write sets conflict
• Undefined in C
  – Divide by zero
  – Shift by negative, shift past bitwidth
  – Signed overflow
  – Etc.
• Solution: Wrap all potentially undefined operations

    int safe_signed_sub (int si1, int si2) {
        if (((si1^si2) & (((si1^((si1^si2) & (1 << (sizeof(int)*CHAR_BIT-1))))^si2)) - si2) < 0) {
            return 0;
        } else {
            return si1 - si2;
        }
    }
• **Undefined pointer behaviors...**
  - Using pointer to null
  - Using pointer to out-of-scope data
  - Creating or using an out of bounds pointer
• Solution:
  – Some problems can be avoided using dynamic checks
    • `if (ptr) { ... }`
  – Some problems require static analysis
    • Dereferencing a global pointer that may reference variables on the stack
    • Casting away type qualifier
l_75 = g_20;
for (l_74 = 4; l_74 != 0;
    l_74 -= 5)) {
    int32_t l_81 = 0xD4B686F2L;
    g_20 = func_78(func_10(g_4,
        ((g_20 <= l_85) & (g_20 &&
            g_20)), 0xA49EL), (p_70 <=
        func_52((l_81 <= l_81), g_20)), l_75,
        ((safe_lshift_func_uint64_t_u_u (l_74, l_76)) != (l_86 == 0xF7AF164004C0D6AFLL)));
}
return g_4;
Results
• Mostly, compilers go wrong at higher optimization levels

• But sometimes the compiler is wrong...
  – Only when optimizations are turned off
  – Consistently at all optimization levels
  – Because it was itself miscompiled
  – Because a system library function is wrong
  – Only very rarely
  – About half of the time
Functional Bug 1 – GCC

• Version of GCC that ships with Ubuntu 8.04 for x86 miscompiles:

```c
int foo (void) {
  signed char x = 1;
  unsigned char y = -1;
  return x > y;
}
```

• Correct return value is 0
Functional Bug 2 – Sun CC

```c
uint32_t x;
int32_t bar (void) {
    return 0xF58AAE07L;
}

void foo (void) {
    x = (0x9AE77AB3L || 1) <= bar ();
}
```

- foo() should assign 0 into x, instead assigns 1
- Wrong code generated at all optimization levels!
- Sun has assigned this bug “Priority 4 – Low”
Functional Bug 3 – LLVM-GCC

```c
int32_t x;
void foo (int32_t y) {
    x = 1;
    if (y){ for (;;) x = 1; }
}
```

• Emitted code does not store to x
• **CompCert is a verified compiler**
  – Compiles C to PPC and ARM
  – Produces a formal proof that the compilation was correct

• **We found**
  – 3 bugs in the frontend
  – 3 bugs in the backend
  – 0 bugs in the (verified) middle part
Volatile Variables

- Abstract C machine tells us how many times each variable is read and written during an execution

Volatile Invariant

- For **volatile qualified** variables, the compiler must issue as many loads as there are reads, and as many stores as there are writes
Volatile Results

• We found systematic miscompilation of volatiles!
  – All compilers have bugs
  – Some are very, very wrong

• What’s going on?
  – Hard to test
  – Volatile conflicts with optimizations
Can We Improve LLVM?

• Over a year we reported 55 bugs to the LLVM developers

• They fixed these bugs and we measured the effect on the quality of this compiler
Compiler crashes

- 13.8% in version 2.0
- 14.6% in version 2.1
- 24% in version 2.2
- 0.0475% in version 2.3
- 0.00103% in version 2.4

6 crash bugs reported between 2.2 and 2.3
19 crash bugs reported between 2.3 and 2.4
Volatile errors

Volatile errors (%) for different LLVM versions:

- LLVM 2.0: 14.8%
- LLVM 2.1: 14.5%
- LLVM 2.2: 12.5%
- LLVM 2.3: 7.47%
- LLVM 2.4: 0%

Key points:
- 4 volatile bugs reported between versions 2.2 and 2.3.
- 5 volatile bugs reported between versions 2.3 and 2.4.
Functional Errors

LLVM Version

0.254% 0.133% 0.127% 0.00723% 0%

5 functional bugs reported between 2.2 and 2.3
16 functional bugs reported between 2.3 and 2.4
LLVM Non-Result #1

• Correlation between our bug reports and compiler quality is obvious

• Causation very hard to prove
  – LLVM team fixed many bugs besides ones that we reported
LLVM Non-Result #2

• Of course LLVM is not now free of bugs

• But it is better when...
  — Compiling the subset of C that we generate
  — Targeting x86
  — Using the standard –O[0123s] options
What If You Find a Compiler Bug?

1. Be extremely suspicious
   – Most suspected compiler bugs turn out to be problems in the compiled code

2. Create a small test case

3. Figure out what the answer is supposed to be

4. Report it!
• Generating bug-inducing test cases is easy and fast
• Creating actionable bug reports is difficult and slow
  – Creating minimum-sized failure-inducing compiler inputs is very hard
• Delta debugging is obvious way to reduce size of failure-inducing tests
  – Delta debugging == Repeatedly remove part of the program and see if it remains “interesting”

• Works well for compiler crashes

• Works poorly for functional and volatile bugs
• Problem: Throwing away part of a program may introduce undefined behavior

• Example:

```c
int foo (void) {
    int x;
    // Error: Invalid code
    return x;
}
```

Oops!
• **Solution 1:** Use the test case generator to reduce program size
  – Generator already knows how to avoid undefined behavior

• **Solution 2:** Bounded exhaustive testing
  – Generate all programs
  – Test smallest ones first
More Problems...

• Assume an overnight run of our tester found 500 programs that trigger compiler failures
  — Did we just find one compiler bug or 500?
  — If more than one, how to prioritize them?
Ongoing Work

• Testing more compilers
  – Especially those for safety-critical embedded systems

• Bug triage

• Identification of flawed or incomplete bug fixes
Lessons Learned

• Random testing is very powerful

• However
  – Adjusting probabilities is hard
  – Generating expressive output that is still correct is hard
Lessons Learned

• Compilers for embedded systems are often highly buggy
  – Even expensive compilers

• Workstation compilers for major platforms are better
  – But still buggy
More Lessons

• Aggressive optimizations are buggy
  – But most compilers have bugs even with minimal or no optimization

• No need to generate exotic code to find compiler bugs
• We already benchmark compilers for performance
• Why not also have benchmarks for compiler correctness?
• Can bounded exhaustive testing + whitebox techniques be used to get formal guarantees about compiler behavior?
Compiler Certification?

• Currently it consists of things like:
  – Passing test suites
  – Being used for a long time

• These are a bad joke

• Compiler output can be meaningfully certified, but not compilers
  – The CompCert project may change this situation
Conclusions

• C compilers require stress testing
  – Test suites insufficient by far
• Generating conforming test inputs is not totally straightforward
• We can benchmark C compiler quality
Volatile Testing Details
Testing Volatile

• Instrumented execution environments monitor accesses to volatile-qualified locations
  – Valgrind for x86
  – RealView ISS for ARM
  – Avrora for AVR
  – Etc.

• Check for violations of the volatile invariant
C program

Compiler 1

Compiler 2

Compiler 3

...
Volatile Bug #1

const volatile int x;
volatile int y;

void foo(void) {
    for (y=0; y>10; y++)
    {
        int z = x;
    }
}

GCC 4.3.0 / IA32 / -Os

foo:  movl  $0, y
     movl  x, %eax
     jmp   .L3
.L2: movl  y, %eax
     incl  %eax
     movl  %eax, y
.L3: movl  y, %eax
     cmpl  $10, %eax
     jg    .L3
     ret
Volatile Bug #2

```c
volatile int a;

void baz(void) {
    int i;
    for (i=0; i<3; i++) {
        a += 7;
    }
}
```

```assembly
baz:
    movl  a, %eax
    leal  7(%eax), %ecx
    movl  %ecx, a
    leal  14(%eax), %ecx
    movl  %ecx, a
    addl  $21, %eax
    movl  %eax, a
    ret
```

LLVM-GCC 2.2 / IA32 / -O2
Do Volatile Bugs Matter?

• A researcher was compiling Linux kernel using LLVM
  – Kernels failed to run – too many accesses to volatiles were optimized away
  – Developers had to manually wrap these accesses in memory barriers

• After 9 volatile bugs that we reported were fixed, compiled Linux kernels run reliably
Why is volatile miscompiled?

• Conflicts with optimizations
• Hard to test
• Compiler test suites don’t contain a lot of volatiles
Experiment 1: Work Around Volatile Errors

• Idea: “protect” volatile accesses from overeager compilers via helper functions

```c
int vol_read_int(volatile int *vp) {
    return *vp;
}

volatile int *vol_id_int(volatile int *vp) {
    return vp;
}

x = vol_1;
vol_1 = 0;
```

```c
x = vol_read_int(vol_1);
*vol_id_int(&vol_1) = 0;
```
## Volatile Helper Results

<table>
<thead>
<tr>
<th>arch. / compiler</th>
<th>vers.</th>
<th>volatile errs. (%)</th>
<th>vol. errs. w/help (%)</th>
<th>vol. errs. fixed (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32 / GCC</td>
<td>3.4.6</td>
<td>1.228</td>
<td>0.300</td>
<td>76</td>
</tr>
<tr>
<td>IA32 / GCC</td>
<td>4.0.4</td>
<td>0.038</td>
<td>0.018</td>
<td>51</td>
</tr>
<tr>
<td>IA32 / GCC</td>
<td>4.1.2</td>
<td>0.195</td>
<td>0.016</td>
<td>92</td>
</tr>
<tr>
<td>IA32 / GCC</td>
<td>4.2.4</td>
<td>0.766</td>
<td>0.002</td>
<td>100</td>
</tr>
<tr>
<td>IA32 / GCC</td>
<td>4.3.1</td>
<td>0.709</td>
<td>0.000</td>
<td>100</td>
</tr>
<tr>
<td>IA32 / LLVM-GCC</td>
<td>2.2</td>
<td>18.720</td>
<td>0.047</td>
<td>100</td>
</tr>
<tr>
<td>AVR / GCC</td>
<td>3.4.3</td>
<td>1.928</td>
<td>0.434</td>
<td>77</td>
</tr>
<tr>
<td>AVR / GCC</td>
<td>4.1.2</td>
<td>0.037</td>
<td>0.033</td>
<td>10</td>
</tr>
<tr>
<td>AVR / GCC</td>
<td>4.2.2</td>
<td>0.727</td>
<td>0.021</td>
<td>97</td>
</tr>
</tbody>
</table>
Why do helpers work?

• Our guess: The rules for volatile accesses are more like function calls than they are like regular variable accesses

• And compilers can get function calls right (usually)
Why do helpers not work?

• Our guess: Compilers were generating wrong code irrespective of volatile
Recommendations

• If you use volatile:
  – Definitely: Look at the compiler output
  – Maybe: Develop test cases for your compiler that come from your code
  – Maybe: Factor volatile accesses into helper functions
  – Maybe: Compile modules that use volatile without optimizations