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?

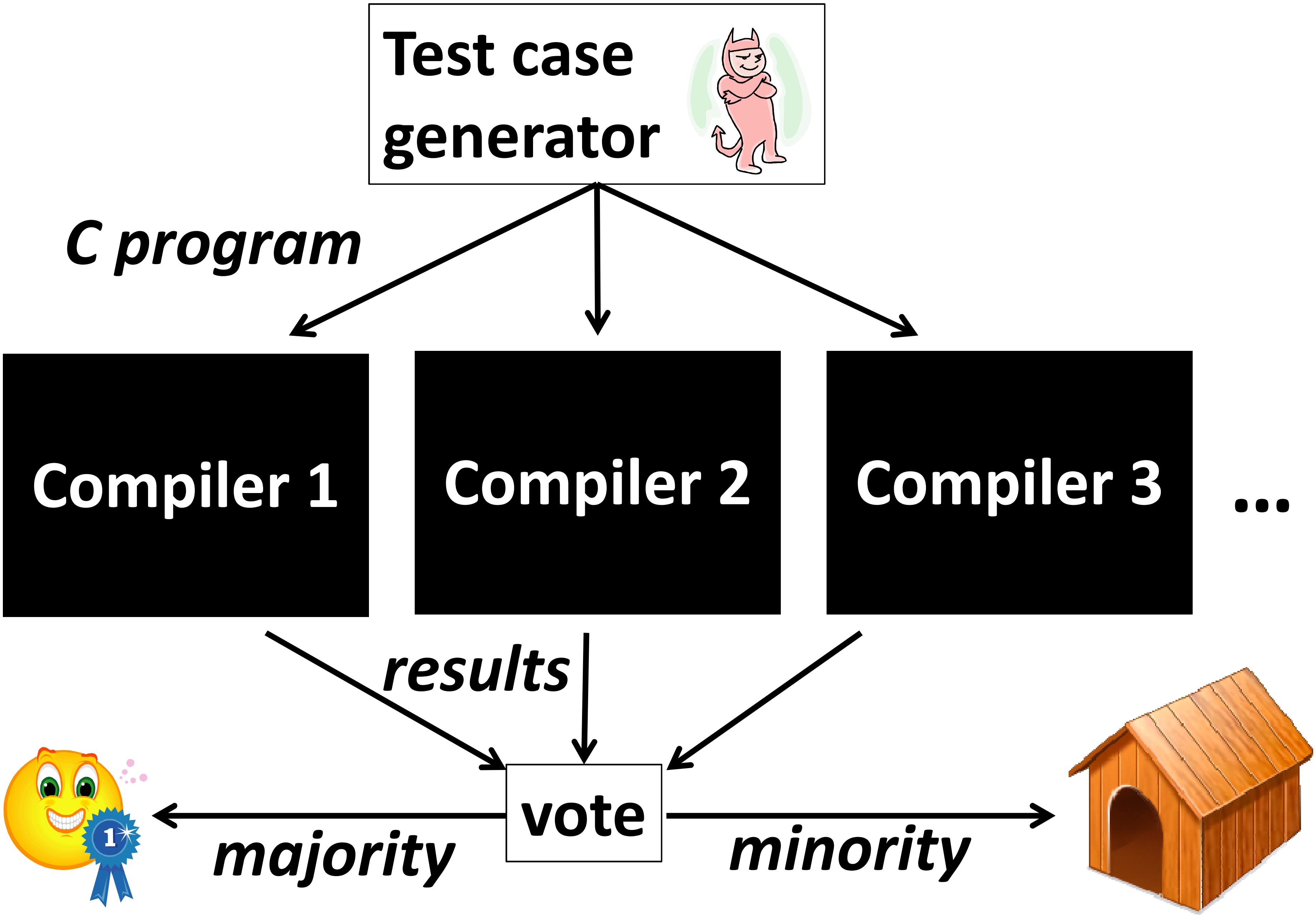
```c
static int x;
static int *volatile z = &x;
static int foo (int *y) {
  return *y;
}
int main (void) {
  *z = 1;
  printf ("%d\n", foo(&x));
  return 0;
}
```

- **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
...
results
vote
majority
minority

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

Not a Bug #1

```
int foo (int x)
{
  return (x+1) > x;
}

int main (void)
{
  printf ("%d\n",
          foo (INT_MAX));
  return 0;
}
```

```
$ gcc -O1 int.c -o int
$ ./int
0
$ gcc -O2 int.c -o int
$ ./int
1
```

Not a Bug #2

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

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

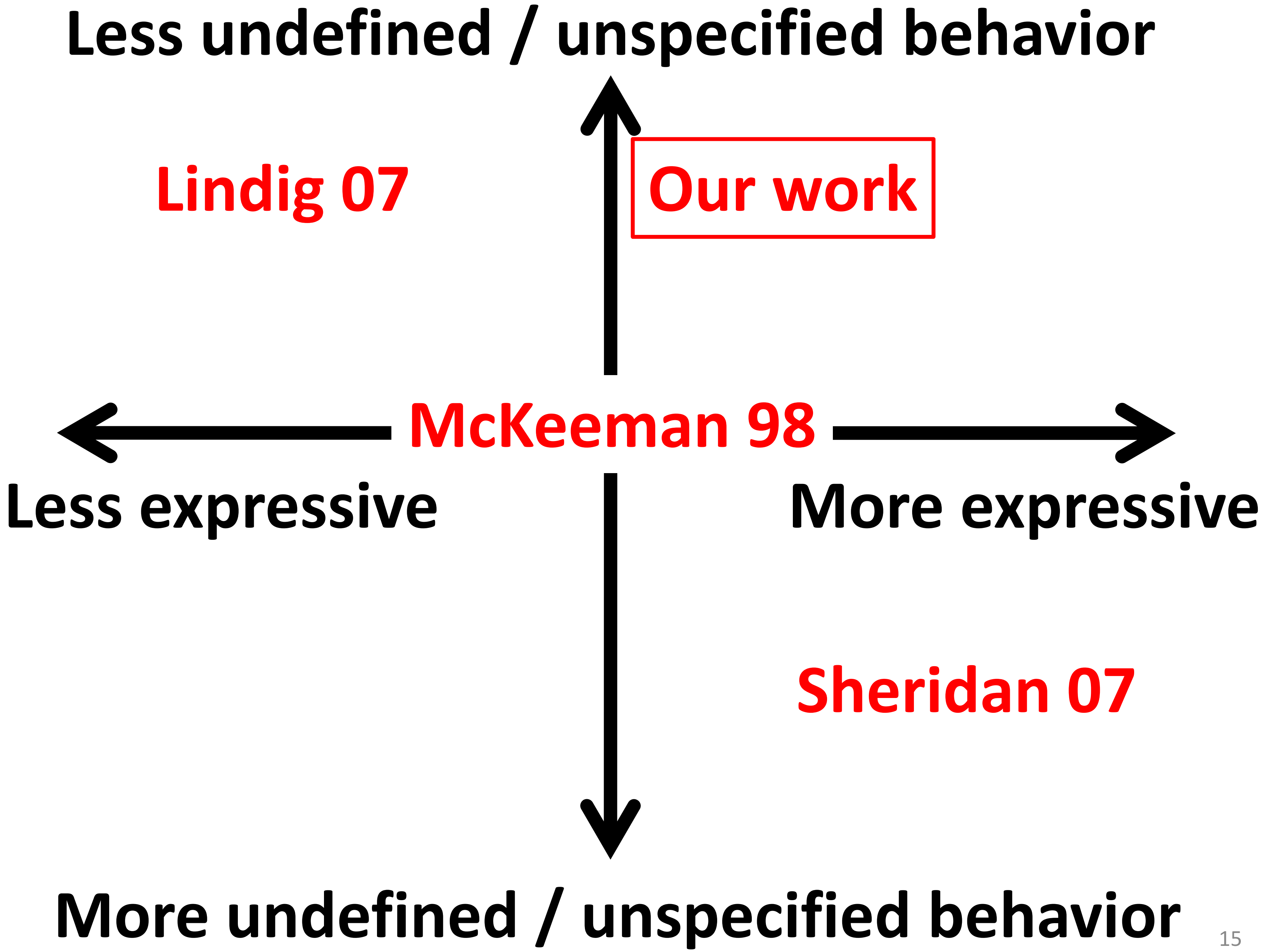
```
$ clang -O0 init.c -o init
$ ./init
51
$ clang -O1 init.c -o init
$ ./init
50
```

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.

```
foo(bar(),baz())
```

- Where bar() and 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**

Integer Undefined Behaviors

- **Undefined in C**
  - Divide by zero
  - Shift by negative, shift past bitwidth
  - Signed overflow
  - Etc.

Undefined Integer Behaviors

- **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;
  }
}
```

Pointer Problems

- **Undefined pointer behaviors…**
  - **Using pointer to null**
  - **Using pointer to out-of-scope data**
  - **Creating or using an out of bounds pointer**

Pointer Problems

- **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:

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

- Correct return value is 0

Functional Bug 2 – Sun CC

```
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

```
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

- Abs ow ma ead and ion

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

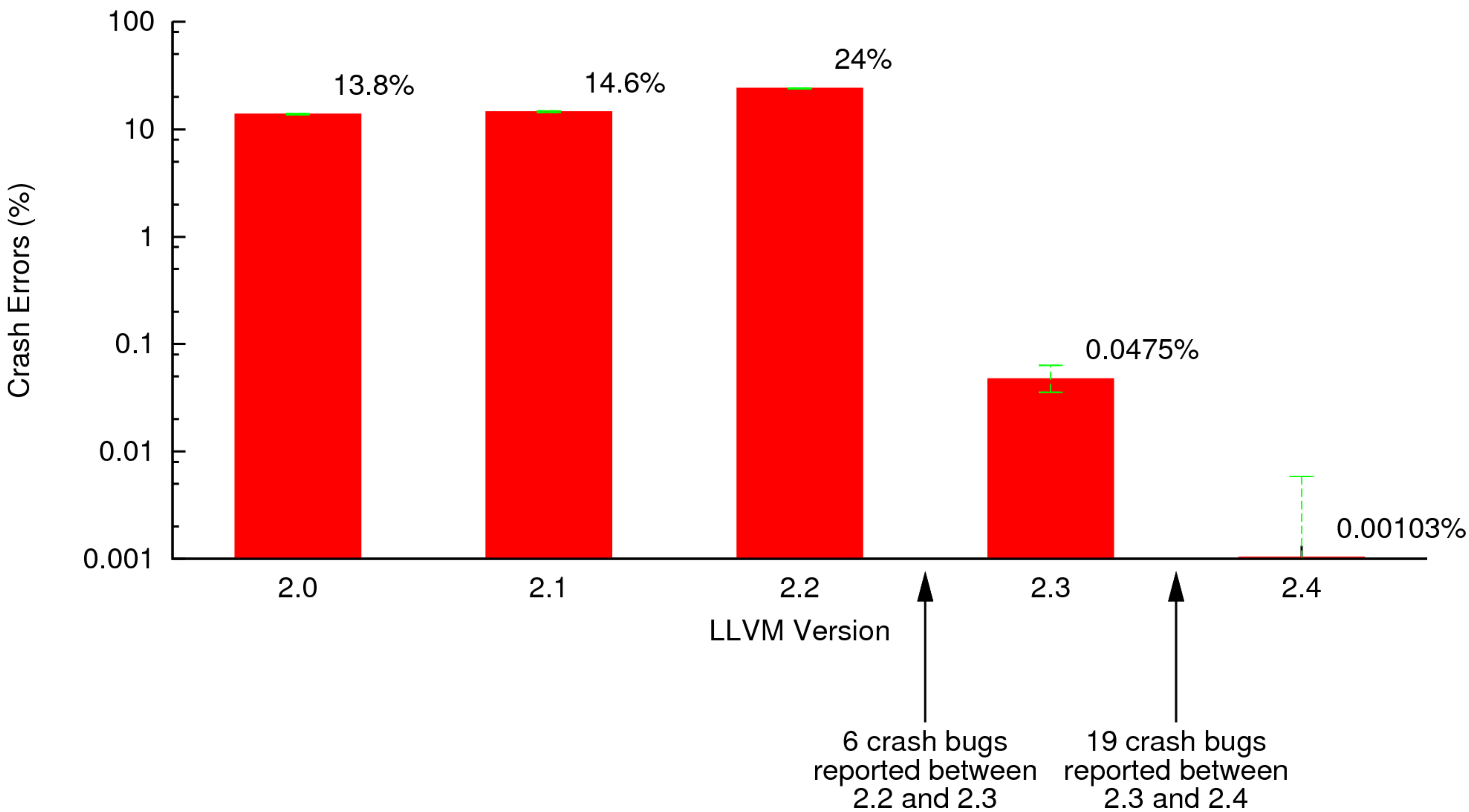
Volatile Results

- **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**

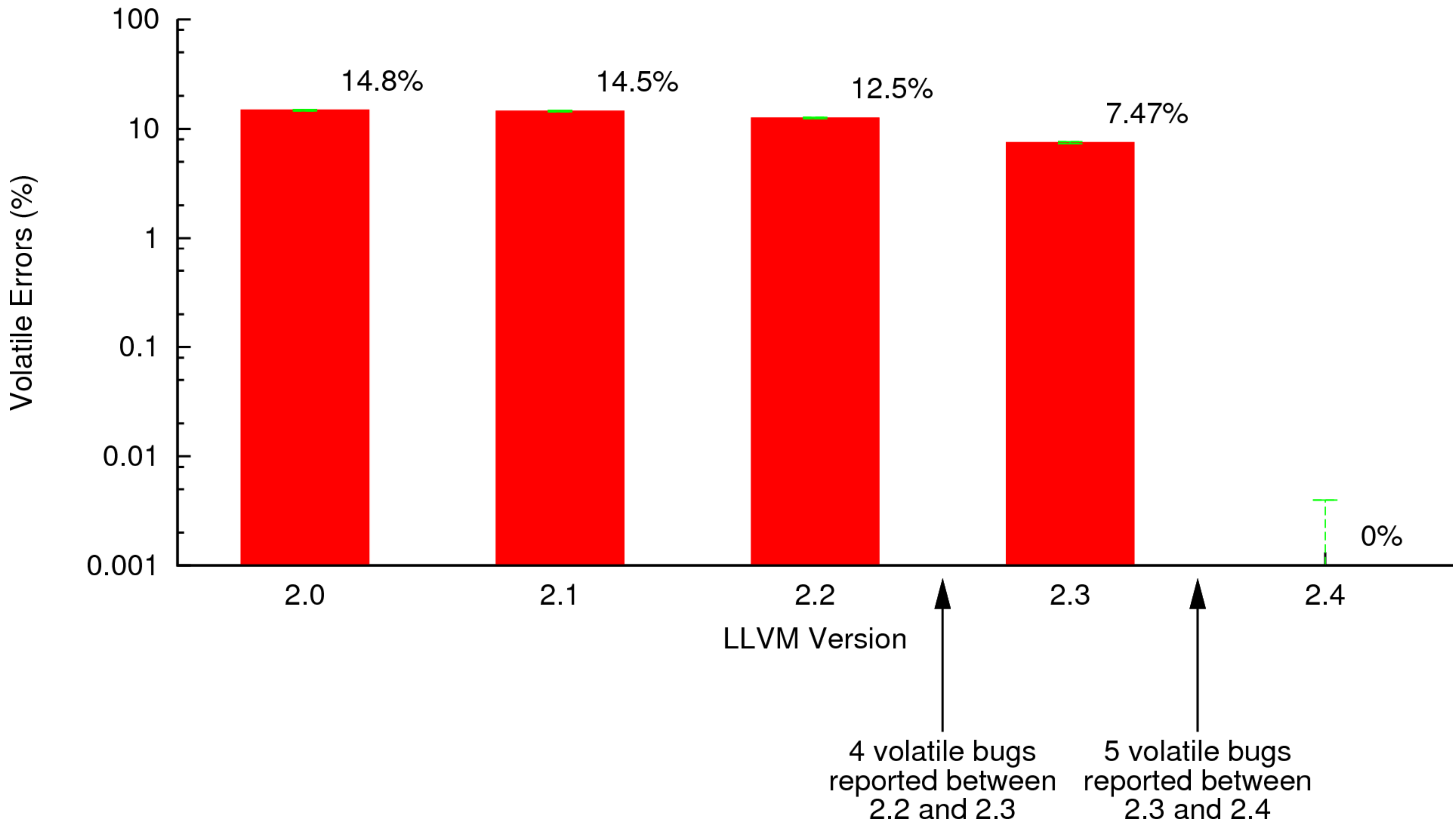
Can We Improve LLVM?

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

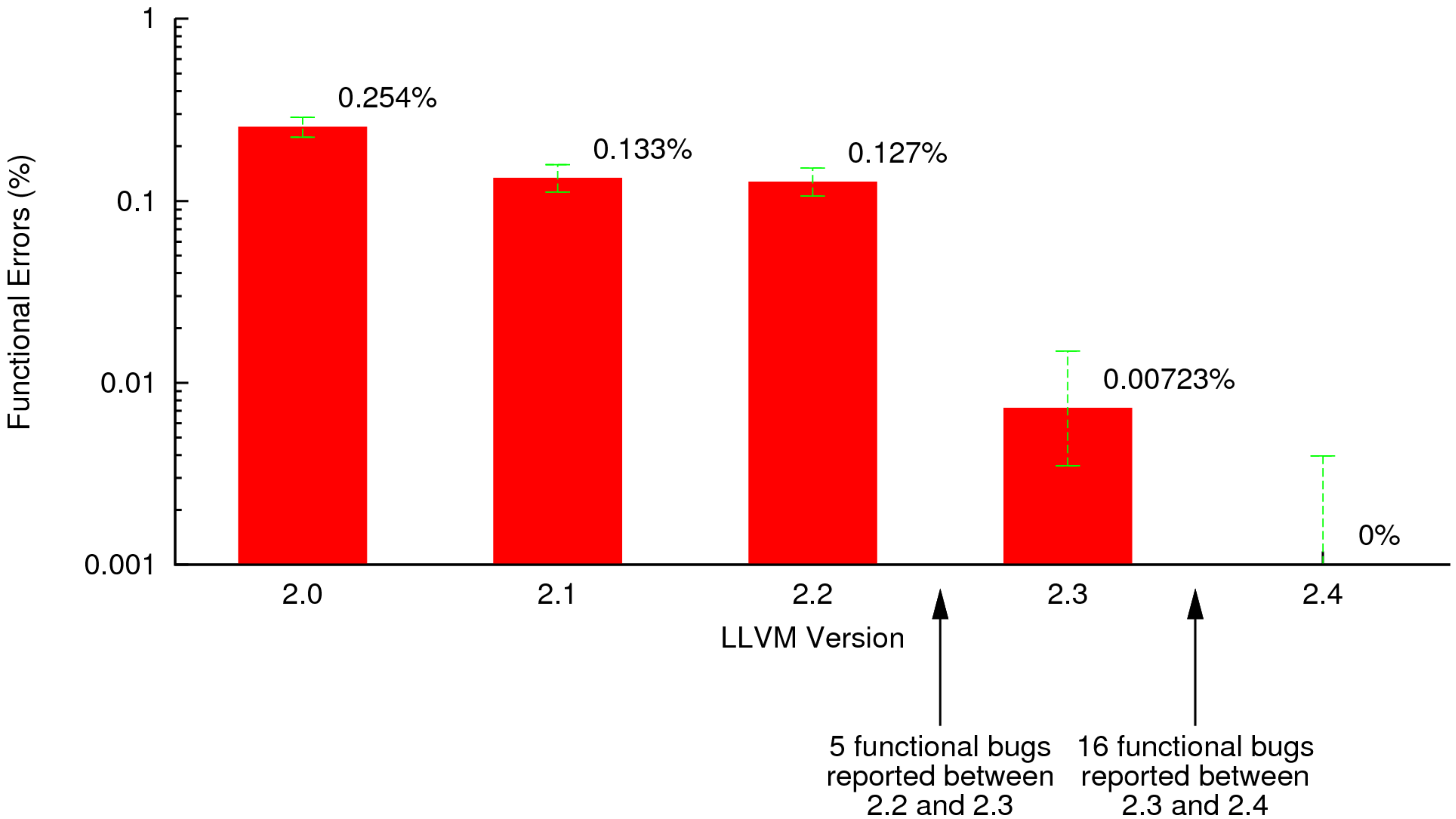
- They fixed these bugs and we measured the effect on the quality of this compiler



Compiler crashes



Volatile errors



Functional Errors

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:**

```
int foo (void) {
  int x;
  x [illegible];
  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

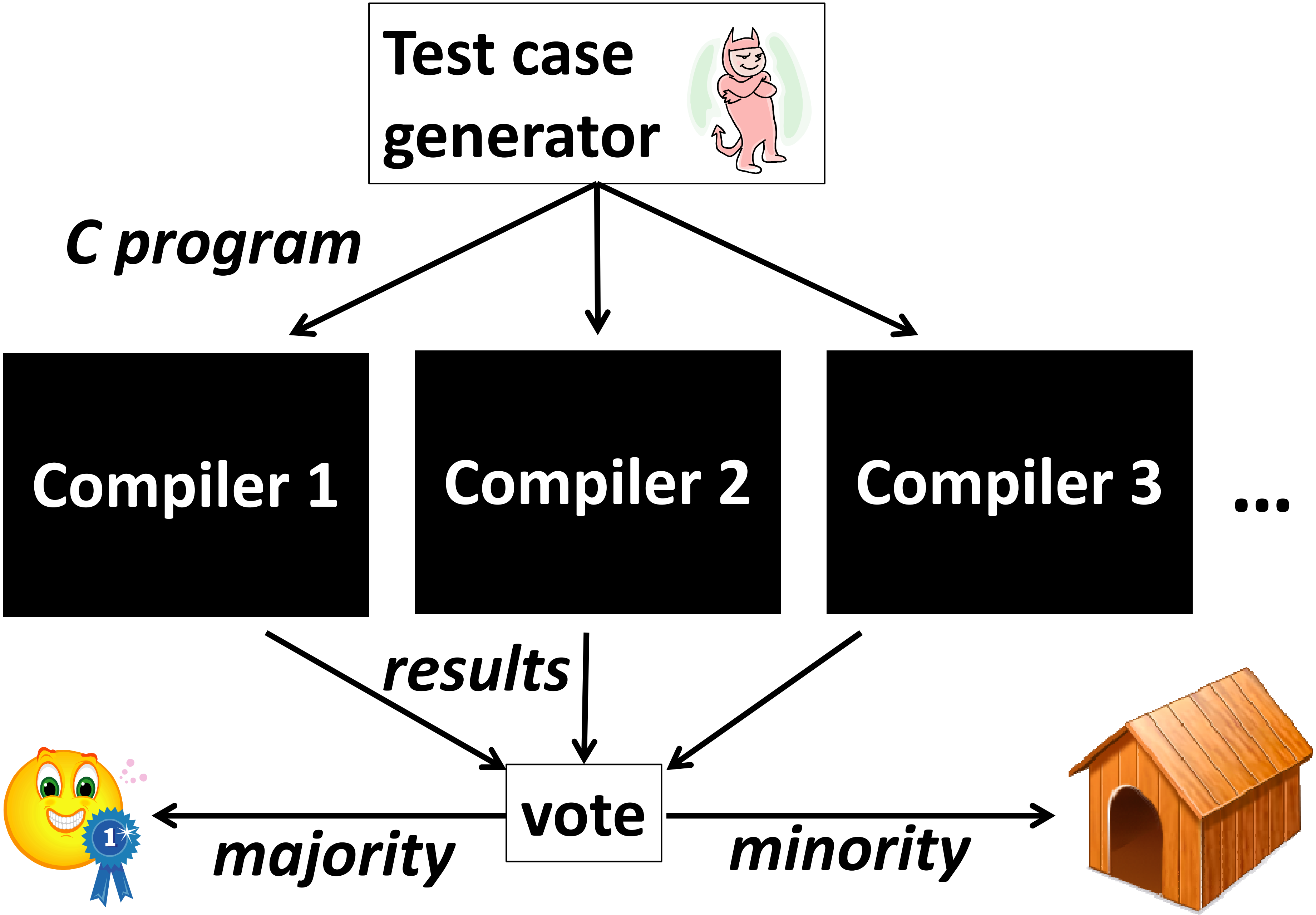
Conclusions

- **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**


Test case generator
C program
Compiler 1
Compiler 2
Compiler 3
...
results
vote
majority
minority

Volatile Bug #1

GCC 4.3.0 / IA32 / -Os

```
const volatile int x;
volatile int y;

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

```
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

```
volatile int a;

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

```
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 hel[per]

opaque

```
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;
```

→

```
x = vol_read_int(vol_1);
*vol_id_int(&vol_1) = 0;
```

Volatile Helper Results

arch. / compiler	vers.	volatile errs. (%)	vol. errs. w/help (%)	vol. errs. fixed (%)
IA32 / GCC	3.4.6	1.228	0.300	76
IA32 / GCC	4.0.4	0.038	0.018	51
IA32 / GCC	4.1.2	0.195	0.016	**92**
IA32 / GCC	4.2.4	0.766	0.002	**100**
IA32 / GCC	4.3.1	0.709	0.000	**100**
IA32 / LLVM-GCC	2.2	18.720	0.047	**100**
AVR / GCC	3.4.3	1.928	0.434	77
AVR / GCC	4.1.2	0.037	0.033	10
AVR / GCC	4.2.2	0.727	0.021	**97**

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