CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
TESTING COMPILERS
Today’s Topic

The compilers are software written by humans, and thus can have errors.

The subtle errors in compilers are especially critical as the compiled programs would behave strangely without a good reason?

How do we find errors or make sure they are absent?
How common are these bugs?

**LLVM**

<table>
<thead>
<tr>
<th>ID</th>
<th>Product</th>
<th>Comp</th>
<th>Assignee</th>
<th>Status</th>
<th>Resolution</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>34376</td>
<td>Build sc</td>
<td>cmake</td>
<td>unassignedbugs</td>
<td>NEW</td>
<td>---</td>
<td>Trying to find __atomic_fetch_add_4 fails on MacOSX</td>
</tr>
<tr>
<td>35935</td>
<td>clang</td>
<td>C++</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>tooling::ToolInvocation catastrophic failure if not -fno-syntax-only</td>
</tr>
<tr>
<td>18276</td>
<td>clang</td>
<td>LLVM Cod</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>clang breaks OpenSSL library when compiling with -O2 or -O3</td>
</tr>
<tr>
<td>26463</td>
<td>clang</td>
<td>C++</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>Run-time assert while building Chromium</td>
</tr>
<tr>
<td>39108</td>
<td>clang</td>
<td>C++</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>constexpr std::string_view global no longer compiles</td>
</tr>
<tr>
<td>35160</td>
<td>tools</td>
<td>lto</td>
<td>unassignedbugs</td>
<td>NEW</td>
<td>---</td>
<td>llvm-dsymutil fails with 'LLVM ERROR: inconsistency in registered CommandLine options' when built with LLVM LINK_LLVM_DYLIB=ON</td>
</tr>
<tr>
<td>24770</td>
<td>clang</td>
<td>C++11</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>Android Native Compile release fails</td>
</tr>
<tr>
<td>31072</td>
<td>clang</td>
<td>C++11</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>clang 3.9 crashes on valid C++11 code on VS 2015, Intel 17 and gcc 4.8</td>
</tr>
<tr>
<td>20738</td>
<td>clang</td>
<td>OpenCL</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>Radeon: code generation for GCN GPUs seems to be broken. CTest benchmark outputs many errors (&quot;Can't spill VOPR!&quot;) and then GPU hangs.</td>
</tr>
<tr>
<td>37903</td>
<td>clang</td>
<td>C++’17</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>Clang/LLVM 7.0.1 on Windows initializes inline static data member multiple times</td>
</tr>
<tr>
<td>26389</td>
<td>clang</td>
<td>LLVM Cod</td>
<td>unassignedclangbugs</td>
<td>REOP</td>
<td>---</td>
<td>[x86-64] clang generate wrong instruction for cygwin</td>
</tr>
<tr>
<td>23783</td>
<td>Build sc</td>
<td>cmake</td>
<td>unassignedbugs</td>
<td>NEW</td>
<td>---</td>
<td>Unable to run &quot;check-all&quot; target when building LLVM 3.6.1</td>
</tr>
<tr>
<td>33435</td>
<td>clang</td>
<td>C++11</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>template template argument bad deduction</td>
</tr>
<tr>
<td>35100</td>
<td>clang</td>
<td>Formatte</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>ClangFormat extension does not work in Visual Studio 2013</td>
</tr>
<tr>
<td>40680</td>
<td>clang</td>
<td>-New Bug</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>clang: error: unable to execute command: Segmentation fault when compiling Apache Oid Dispatch</td>
</tr>
<tr>
<td>20326</td>
<td>clang</td>
<td>Headers</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>recent header breakage in clang</td>
</tr>
<tr>
<td>39820</td>
<td>clang</td>
<td>-New Bug</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>clang report header file not found when using -E but file exists</td>
</tr>
<tr>
<td>29159</td>
<td>new-bugs new-bugs</td>
<td>new-bugs</td>
<td>unassignedclangbugs</td>
<td>NEW</td>
<td>---</td>
<td>reproducible assert with -Wdocumentation with clang r279749 with reduce'd test case</td>
</tr>
<tr>
<td>34121</td>
<td>libranir</td>
<td>Backend:</td>
<td>unassignedbugs</td>
<td>NEW</td>
<td>---</td>
<td>r4 writes inserted overwriting a register that is supposed to be preserved across function calls</td>
</tr>
<tr>
<td>30910</td>
<td>libranir</td>
<td>MCIJT</td>
<td>unassignedbugs</td>
<td>NEW</td>
<td>---</td>
<td>Deleting MCIJIT execution engine causes SIGSEGV. Segmentation fault in libstdc++ exceptions handling (_cxa_throw)</td>
</tr>
</tbody>
</table>
How common are these bugs?

This list is too long for Bugzilla's little mind; the Next/Prev/First/Last buttons won't appear on individual bugs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Product</th>
<th>Comp</th>
<th>Assignee</th>
<th>Status</th>
<th>Resolution</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>24591</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>poor diagnostic with a missing \ in a class definition</td>
</tr>
<tr>
<td>22431</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>- Warns about missing usage of const initializer list in synthesized clors</td>
</tr>
<tr>
<td>21139</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Improve handling of function-scope statics on platforms without weak symbols</td>
</tr>
<tr>
<td>18638</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>macros should be expanded in #pragma align for C++</td>
</tr>
<tr>
<td>24847</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Instantiates un-called copy constructor</td>
</tr>
<tr>
<td>25322</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>ISO compliance of defining structs in anonymous unions</td>
</tr>
<tr>
<td>18069</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Contradicting type and variable attributes</td>
</tr>
<tr>
<td>19502</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>duplicate diagnostic for invalid template constant parameter</td>
</tr>
<tr>
<td>19965</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Invalid member declaration diagnosed late</td>
</tr>
<tr>
<td>29556</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Expect error when 'using namespace std' is declared when std namespace is not defined</td>
</tr>
<tr>
<td>30060</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Error/warning on invalid code (duplicate identifier for enum/class) should be more specific</td>
</tr>
<tr>
<td>20040</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>A new expression must check the access level of delete operator</td>
</tr>
<tr>
<td>23263</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Incomprehensible message for invalid attempt to partially specialize a member</td>
</tr>
<tr>
<td>31326</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>data members in multiple inheritance</td>
</tr>
<tr>
<td>17000</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>parse error: calling member template function of non-value from within template class member</td>
</tr>
<tr>
<td>32143</td>
<td>gcc</td>
<td>c++</td>
<td>UNCO</td>
<td>NEW</td>
<td>---</td>
<td>declrtl generated with incorrect visibility</td>
</tr>
<tr>
<td>19501</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>Redundant &quot;template&quot; keyword rejected</td>
</tr>
<tr>
<td>31164</td>
<td>gcc</td>
<td>c++</td>
<td>UNCO</td>
<td>NEW</td>
<td>---</td>
<td>Problem with GCC 4.1 and Boost signals</td>
</tr>
<tr>
<td>15269</td>
<td>gcc</td>
<td>c++</td>
<td>unassigned</td>
<td>NEW</td>
<td>---</td>
<td>attribute ((deprecated)) broken with inline, ignored with pure virtual, misreported after definition</td>
</tr>
</tbody>
</table>
How do the programs fail?

The compiler crashes, or gives an assertion failure

(I bet you’ve seen those in your MPI)
How do the programs fail?

PHP version 5.3.4, the interpreter enters an infinite loop:

```php
<?php
    $result = 0; // ...
    $result = $result + 2.2250738585072011e-308;
    printf( "Final result: %.17e \n", $result);
?>

Similar thing happened with Java VM…
How do the programs fail?

The code may be generated just fine, but the optimization introduced an error or assumed something that it shouldn’t have

```c
int foo (void) {
    signed char x = 1;
    unsigned char y = 255;
    return x > y;
}
```

Bug found by CSmith (PLDI’11) in the version of GCC that shipped with Ubuntu 8.04. It compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

Survey of some potential issues with compiler optimizations:
- Dangerous Optimizations and the Loss of Causality
Compiler Fuzzing

Key idea: Generate many programs and compile them. The compiler should still be able to produce (ideally correct) code for these programs.

Questions:
- How to generate programs?
- How to know they are correct?
- How to identify where the error may be?
C Smith

Generates arbitrary C programs that conform to the C99 standard.

- *Finding and Understanding Bugs in C Compilers* Xuejun Yang, Yang Chen, Eric Eide, John Regehr (PLDI ’11)
- Explores atypical combinations of C language features
- Found many bugs in existing compilers
- Key challenge is targeting program generation to more likely reveal potential problems
- Trivia: the program size that helped discover most bugs was around 82KB

Fuzzing (for various purposes) is a vibrant research area these days
Checking for Correctness

1. Make sure the compiler is not behaving unexpectedly: crashing, diverging, etc.

2. Compare generated programs:
   - Compile with multiple compilers/versions or optimization levels and see if the results differ (see e.g., CSmith)
   - Change a program in some controlled manner (V. Le at al. PLDI’14, OOPSLA’15)

http://web.cs.ucdavis.edu/~su/emi-project/
Example from EMI (PLDI’14)

```c
struct tiny { char c; char d; char e; }
void foo(struct tiny x) {
    if (x.c != 1) abort();
    if (x.e != 1) abort();
}
int main() {
    struct tiny s;
    s.c = 1; s.d = 1; s.e = 1;
    foo(s);
    return 0;
}
```

A bug in the LLVM optimizer causes this miscompilation. The developers believe that the Global Value Numbering (GVN) optimization turns the struct initialization into a single 32-bit load. Subsequently, the Scalar Replacement of Aggregates (SRoA) optimization decides that the 32-bit load is undefined behavior, as it reads past the end of the struct, and thus does not emit the correct instructions to initialize the struct. The developer who fixed the issue characterized it as

"... very, very concerning when I got to the root cause, and very annoying to fix."

Figure 2: Reduced version of the code in Figure 1b for bug reporting. (http://llvm.org/bugs/show_bug.cgi?id=14972)

```c
int a, b, c, d, e;
int main() {
    for (b = 4; b > -30; b--)
        for (; c;)
            for (;;) {
                e = a > 2147483647 - b;
                if (d) break;
            }
    return 0;
}
```

Partial Redundancy Elimination (PRE) detects the expression “e2147483647 - b” as loop invariant. Loop Invariant Motion (LIM) tries to move it up from the innermost loop to the body of the outermost loop. Unfortunately, this optimization is problematic, as GCC then detects a signed overflow in the program’s optimized version and this (incorrect) belief of the existence of undefined behavior causes the compiler to generate non-terminating code (and the bogus warning at -02).

Figure 3: GCC miscompiles this program to an infinite loop instead of immediately terminating with no output. (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58731)
Test Oracles

1. Detect crashing or hanging compilers

2. Differential Testing
   • Try multiple compilers – do they result in the same outcome
   • Cross-compiler, Cross-optimization, Cross-version
   • Compare the results of the compiled program or the code itself

3. Metamorphic testing
   • Change the input program in a way that you expect the output to change.
   • E.g., if the program compiles \( f(x) \) \{ return \( x \) \} then it should compile \( g(x) \) \{ return \( 2\times x \) \}
   • But more often, these relations are equivalences, even for a particular input, e.g., \( f(x) \) \{ return \( x + 2 \) \} and \( g() \) \{ return \( 4 \) \} are equivalent when testing \( f(2) \) and \( g() \).
### Example Metamorphic Relations

<table>
<thead>
<tr>
<th>Paper</th>
<th>Metamorphic relation</th>
<th>How to construct metamorphic relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tao et al. [114]</td>
<td>Equivalence relation</td>
<td>Constructing equivalent expressions, assignment blocks, and submodules</td>
</tr>
<tr>
<td>Le et al. [72]</td>
<td>Equivalence relation under a given set of test inputs</td>
<td>Deleting code in the dead regions under the set of test inputs</td>
</tr>
<tr>
<td>Le et al. [73]</td>
<td>Equivalence relation under a given set of test inputs</td>
<td>Deleting and inserting code in the dead regions under the set of test inputs</td>
</tr>
<tr>
<td>Sun et al. [111]</td>
<td>Equivalence relation under a given set of test inputs</td>
<td>Inserting code in both the live and dead regions by synthesizing valid semantic-preserving code snippets under the set of test inputs</td>
</tr>
<tr>
<td>Donaldson and Lascu [43]</td>
<td>Equivalence relation</td>
<td>Injecting dead code into test programs</td>
</tr>
<tr>
<td>Nakamura and Ishiura [89]</td>
<td>Equivalence relation</td>
<td>Applying a set of equivalent transformation rules on test programs</td>
</tr>
<tr>
<td>Donaldson et al. [42]</td>
<td>Equivalence relation</td>
<td>Applying a set of (essentially) semantics-preserving transformations on high-value graphics shaders</td>
</tr>
<tr>
<td>Samet [99–101]</td>
<td>Equivalence relation</td>
<td>Converting a source program and the object program into an intermediate representation, respectively</td>
</tr>
</tbody>
</table>

*Table from A Survey of Compiler Testing, CSUR February 2020*
Program Reduction

Aid in fault localization by reducing the size of the program that reveals an error.

• Once we know the error, we can start removing program statements for as long as the smaller program still reveals the same error

• See e.g.,
  • CReduce, PLDI’12
  • Perses, ICSE’18
Verification

• Verified compilation: CompCert is a verified, optimizing compiler for a large subset of C
  • [http://compcert.inria.fr/compcert-C.html](http://compcert.inria.fr/compcert-C.html)
  • Backed by powerful solvers
  • The executable code *is formally proved* to behave exactly as specified by the program semantics

• Formalizing semantics: An Executable Formal Semantics of C with Applications; Chucky Ellison and Grigore Rosu (POPL’12)
  • [http://www.kframework.org](http://www.kframework.org)
Performance Measuring

*Not trivial! There are more concerns than just run and collect timings*

Memory layouts can have up to 10% execution time variability:

- Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney, “Producing Wrong Data Without Doing Anything Obviously Wrong!” (ASPLOS’09)

One solution: Stabilizer (ASPLOS’13) for performance measurement with randomized memory layouts:

- [https://emeryberger.com/research/stabilizer/](https://emeryberger.com/research/stabilizer/)

Guidelines for measuring time: Kalibera and Jones, “Rigorous Benchmarking in Reasonable Time” (ISMM’13)
Identify Missed Optimizations

Compare the generated code statically / dynamically


For more details, see

A Survey of Compiler Testing, CSUR February 2020 ()
https://dl.acm.org/doi/10.1145/3363562

Csmith: https://embed.cs.utah.edu/csmith/
Creduce: https://embed.cs.utah.edu/creduce/
EMI: https://web.cs.ucdavis.edu/~su/emi-project/