Volatiles Are Miscompiled, and What to Do about It

Eric Eide
University of Utah, School of Computing
Salt Lake City, UT USA
eeide@cs.utah.edu

John Regehr
University of Utah, School of Computing
Salt Lake City, UT USA
regehr@cs.utah.edu

ABSTRACT

C's volatile qualifier is intended to provide a reliable link between operations at the source-code level and operations at the memory-system level. We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables. This result is disturbing because it implies that embedded software and operating systems—both typically coded in C, both being bases for many mission-critical and safety-critical applications, and both relying on the correct translation of volatiles—may be being miscompiled.

Our contribution is centered on a novel technique for finding volatile bugs and a novel technique for working around them. First, we present *access summary testing*: an efficient, practical, and automatic way to detect code-generation errors related to the volatile qualifier. We have found a number of compiler bugs by performing access summary testing on randomly generated C programs. Some of these bugs have been confirmed and fixed by compiler developers. Second, we present and evaluate a workaround for the compiler defects we discovered. In 96% of the cases in which one of our randomly generated programs is miscompiled, we can cause the faulty C compiler to produce correctly behaving code by applying a straightforward source-level transformation to the test program.

Categories and Subject Descriptors D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.3.2 [**Programming Languages**]: Language Classifications—*C*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms Languages, Reliability

Keywords compiler testing, compiler defect, automated testing, random testing, random program generation, volatile

1. INTRODUCTION

A C program's connection to its underlying memory subsystem is tenuous: optimizing compilers are free (and in fact try very hard) to cache values in registers, to reorder computations, and to eliminate useless and redundant computations. The `volatile` qualifier is intended to provide a reliable anchor between variables and the memory system. Briefly stated, when a storage location is marked as volatile, the C compiler must ensure that every use (read or write) of that location in the source program is realized by an appropriate memory operation (load or store) in the compiled program. Accesses to volatiles are considered to be side-effecting operations, and they are therefore part of the observable behavior of a program that must not be changed by an optimizing compiler. Embedded software commonly relies on volatile variables in order to access memory-mapped I/O ports, to communicate between concurrent threads or processes, and to communicate between interrupt handlers and the main computation.

To see why volatiles are special, notice that memory-mapped I/O registers may have semantics very different from RAM. For example, a register that reflects sensor values may produce a different value each time is it loaded, and may not support stores at all. Additionally, both stores to and loads from a memory-mapped register may be side-effecting operations. Consider, for instance, the following C code for a function that resets a watchdog timer in a hypothetical embedded system:

```
/* linker will map to the proper I/O register */
extern volatile int WATCHDOG;

void reset_watchdog() {
  WATCHDOG = WATCHDOG; /* load, then store */
}
```

This function is called periodically by the embedded control program to signal that it is still running. Watchdog timers are commonly used in safety-critical software as a guard against software faults; if the timer expires, the software must have crashed or become wedged, and a system restart is forced. The action required to reset a watchdog timer is determined by hardware designers. Here we assume that it is a load from a hardware register belonging to the watchdog subsystem, followed by a store to the same register.

Regardless of optimization level, a correct compiler must turn the function above into object code that loads and then stores the `WATCHDOG` register. Recent versions of GCC for IA32 emit correct assembly code:

```
reset_watchdog:
        movl    WATCHDOG, %eax
        movl    %eax, WATCHDOG
        ret
```

On the other hand, the latest version of GCC's port to the MSP430 microcontroller compiles the code into the following assembly:

```
reset_watchdog:
        ret
```


The definitive version was published in *Proceedings of the Eighth ACM and IEEE International Conference on Embedded Software (EMSOFT)*, Atlanta, Georgia, USA, Oct. 2008, `http://doi.acm.org/10.1145/1450058.1450093`

The function is a nop and applications that rely on it to reset the watchdog timer will not work. Although the symptoms of this compiler bug—spurious periodic reboots due to failure to reset the watchdog timer—may be relatively benign, the situation could be worse, for example, if the hardware register were used to lower control rods, cancel a missile launch, or open the pod bay doors. Most compilers based on GCC 3.x, including the version of GCC's port to the AVR microcontroller that is currently used by TinyOS [9], generate similar incorrect output for this trivial example.

We have found that compiler bugs of this type are not the exception: instead, they are disturbingly common. When we first came across these bugs, we created a small suite of test programs similar to the watchdog example above. We found that many compilers produced visibly incorrect object code for one or more of our tests. To investigate the extent of volatile-access problems more thoroughly, we decided to employ random testing. We developed an appropriate C program generator and compiler test harness, and we used them to automatically generate test cases, automatically find bugs in compiling accesses to volatiles, and investigate the possibility of automatically working around compiler defects through program transformation.

The contributions of our paper are the following:

1. We show that problems implementing C's `volatile` qualifier are widespread. All compilers that we examined, including a number of production-quality compilers for embedded systems, produce incorrect object code for at least one input.
2. We present a technique for randomly generating *nearly strictly conforming C programs* that must perform the same computation across a broad class of platforms and compilers.
3. We describe *access summary testing*, our technique for effectively and automatically detecting miscompilation of accesses to volatile variables.
4. We show that in many cases, the impact of compiler bugs can be mitigated by introducing small helper functions into a program. We evaluate the costs of this refactoring.
5. Based on our findings, we provide concrete recommendations for application developers and compiler developers.

2. WHAT DOES VOLATILE MEAN?

We answer this question in two parts. The first provides a practical and intuitive explanation, and is a sufficient basis for understanding the rest of this paper. The second addresses some additional subtleties found in the C standard.

2.1 Practical answer

The proper behavior of a `volatile`-qualified variable is this:

> For every read from or write to a volatile variable that would be performed by a straightforward interpreter for C, exactly one load from or store to the memory location(s) allocated to the variable must be performed.

For example, if a variable `i` is declared as `volatile int`:

- `i++` must result in a load from `i` and then a store to it.
- `(x || i)` must result in a load from `i` iff `x` evaluates to false.
- `*p = 5` must result in a store to `i`, provided that `p` points to `i` and has type `volatile int *`.

A compiler may not move accesses to volatile variables across sequence points.[1] No guarantees are made about the atomicity of any given volatile access, about the ordering of multiple volatile accesses between two consecutive sequence points, or about the ordering of volatile and non-volatile accesses. For example, the following code illustrates a common mistake in which a `volatile` variable is used to signal a condition about a non-volatile data structure, perhaps to another thread:

```
volatile int buffer_ready;
char buffer[BUF_SIZE];

void buffer_init() {
  int i;
  for (i=0; i<BUF_SIZE; i++)
    buffer[i] = 0;
  buffer_ready = 1;
}
```

The for-loop does not access any volatile locations, nor does it perform any side-effecting operations. Therefore, the compiler is free to move the loop below the store to `buffer_ready`, defeating the developer's intent. Making the buffer volatile would prevent this transformation. (A better solution would be to avoid using `volatile` to implement inter-thread communication, and instead to use synchronization primitives that contain compiler and hardware memory barriers.)

Although a bit nonintuitive, variables qualified as both `const` and `volatile` make sense and are useful. In a system where a communication variable is written by thread *A* and read by thread *B*, thread *B*'s code might mark the variable as `const volatile` in order to turn accidental writes into compile-time errors. Similarly, `const volatile` is a good model for a hardware register whose value may change unpredictably, but that should not be written to.

2.2 Subtleties of the C standard

In Section 6.7.3 the C99 standard [7] says:

> An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, ... Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously. What constitutes an access to an object that has volatile-qualified type is implementation-defined.

A footnote in the same section elaborates:

> A `volatile` declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

Although the intended semantics of `volatile` are clear, the language in the standard is vague on two important points (neither of which is a concern for this paper).

[1] According to Section 3.8 of the C FAQ [18], "A sequence point is a point in time at which the dust has settled and all side effects which have been seen so far are guaranteed to be complete. The sequence points listed in the C standard are at the end of the evaluation of a full expression (a full expression is an expression statement, or any other expression which is not a subexpression within any larger expression); at the `||`, `&&`, `?:`, and comma operators; and at a function call (after the evaluation of all the arguments, and just before the actual call)."

First, it is not apparent to what extent—if any—the compiler must use memory barriers, non-cacheable memory regions, or similar mechanisms to ensure that accesses to volatile objects are committed to RAM in the specified order. A strict reading of the standard would seem to require the insertion of memory barriers at sequence points before and after volatile accesses, but in practice few compilers do this. Rather, a programmer must manually insert the necessary hardware operations.

The second problem is the language "what constitutes an access... is implementation-defined." This part of the standard admits the possibility of a conforming C implementation that entirely ignores the `volatile` qualifier (provided that this is documented). However, such an implementation would be utterly useless for systems programming and we will not consider this possibility further. Based on a USENET post from one of the C standards committee members [5], it seems that the offending sentence is a poorly worded reflection of the fact that some hardware platforms have a minimum access width. For example, on a machine that supports only 32-bit memory operations, an access to a volatile byte will unavoidably access the other three bytes in the same word, even if one or more of these is also volatile. Modern computer architectures generally support memory operations at byte granularity.

3. GENERATING PROGRAMS TO TEST VOLATILE CORRECTNESS

By writing simple tests by hand, we discovered that many C compilers have bugs when it comes to implementing the semantics of volatile variables. Hand-written tests, however, are inherently limited in both number and complexity. *Random testing*, on the other hand, uses automatically generated inputs to test a piece of software. Therefore, to "scale up" our discovery of bugs, we decided to use a random program generator to find problems in the ways that C compilers handle volatile variables.

The key question that needs to be answered before using a random program generator to find compiler bugs is: *How can we generate programs in a sufficiently constrained way that their behavior with respect to volatile objects must be the same across all compilers and sets of compiler options?* The solution is to generate programs that avoid behaviors that the C standard leaves unspecified, undefined, or implementation-defined. As we explain below, this required a surprising amount of work and involves both compile-time and run-time restrictions on program behavior.

3.1 Overview of randprog

Our random C program generator is called *randprog*; it is a significantly enhanced and tailored version of a program generator written by Bryan Turner [21]. The output of our new generator is a random C program that manipulates volatile variables. In other words, every time *randprog* is run, it produces a (probably) unique C program. Each of these is a potential test case for the C compilers that we study. By running *randprog* over and over, we can quickly produce thousands of test cases. Furthermore, we can script the production and evaluation of test cases, yielding a fully automatic procedure to search for compiler bugs.

randprog creates C programs that perform computations over signed and unsigned integer variables with various ranges: 8, 16, and 32 bits. (Our current tests do not use characters, arrays, pointers, structures, unions, or floating-point values.) Within these limits, our goal is to generate programs with a wide range of behaviors and characteristics. Each generated program has three parts:

- The first is a set of randomly generated global variables. Each is explicitly initialized to a constant value (randomly chosen by *randprog*). Some of the globals may be declared `const`, some may be declared `volatile`, and some may be `const volatile`.
- The second is a set of randomly created functions. Each accepts some number of (integer-typed) arguments and returns an integer value. Each function body contains local (non-volatile) variable declarations, assignments, if-then-else statements, for-loops, function invocations, and returns. Expressions are made from accesses to global and local variables; uses of built-in arithmetic, logical, and bitwise operators; and invocations of other generated functions. The randomly generated expressions can be quite complicated. However, assignment operators occur only at statement level and within the initialization and increment parts of for-loops.

  The generated functions may be mutually recursive. *randprog* does not guarantee that functions will terminate, but in fact, most generated programs do terminate in a short amount of time.
- The third part is a small amount of runtime support, including the program's `main` function. The task of the main function is to invoke the "topmost" randomly generated function and then compute a checksum as described below.

The C programs created by *randprog* are both *closed* and *nearly strictly conforming*. These two properties allow us to determine the expected outcome of a test case, as we describe in Section 4. A closed program is simply one that takes no inputs; all of the program data is contained within the program source code. It is simple to ensure that the programs created by *randprog* satisfy this property. A strictly conforming program, as defined by the C language standard, is a program whose output does not depend on any unspecified, undefined, or implementation-defined behavior.[2] Ensuring this property of our generated programs requires some care within the generator, and it is (almost completely) enforced using a combination of code-generation constraints and runtime support.

3.2 "Nearly strictly conforming"

To understand why our random programs are *nearly* strictly conforming, note that "strictly conforming" is a property over a program's output. The programs created by *randprog* have two kinds of output. The first is a checksum over the program's global variables, which is computed just before the program terminates. Depending on the target computing platform, the program either prints the checksum to standard output or stores it into a well-known memory location. The second output is a sequence of reads and writes to the program's volatile variables. This sequence is an output because—in addition to being the behavior we study in this work—accesses to volatile variables are defined by the C standard to be side-effecting operations.

As described more fully in Section 4, for each test program, we judge the correctness of a compiler by executing the compiled version of the test and examining the test program's outputs. We then need to decide if the outputs are right or wrong. To make this decision tractable, *it is important that the outputs of the test program be well defined.* In particular, the outputs must not depend on the compiler, compiler options, or hardware platform in use. If a generated program performs operations whose results are unspecified, undefined, or implementation-defined, the data calculated by

[2] An *unspecified behavior* is one for which the C standard allows multiple possibilities without restricting how the choice is made. An example is the order of evaluation for function arguments, which may vary across compilers, across compiler options, or even across the function calls within a single program. An *implementation-defined behavior* is an unspecified behavior, where each compiler must document the behavior that it actually implements. An *undefined behavior* is the outcome of a nonportable construct or a programming error, such as division by zero. The standard imposes no requirements on undefined behaviors.

the program may vary—based on numerous factors—thus leading to different executed paths and ultimately to varying program outputs. In the rest of this section, we detail how *randprog* produces programs that behave consistently (or else reveal compiler bugs!) across all our tested compilers and platforms.

Static assurances. We have designed *randprog* so that many potential sources of inconsistent behavior are eliminated statically, i.e., by the nature of the code that is output by the code generator. For example, our generated programs:

- use integer types of defined size (e.g., `int32_t`), not integers with implementation-defined sizes (e.g., `int`);
- always initialize local variables; and
- always specify a return value (i.e., `return` *`expr`*) when returning from a function with a non-void return type.

Dynamic checks. Many not-well-defined behaviors, however, are difficult to avoid statically because they depend on the dynamic (run-time) values within the program. For example, if `i` is a signed integer variable that currently holds a negative value, then the result of `(i >> 1)` is undefined by the C standard. The result *is* defined, however, when `i` holds a non-negative value.

Therefore, we rely on dynamic checks in our generated programs to avoid operations that would (potentially) affect the data values that are computed. These checks are implemented by functions in a small runtime library, which *randprog* includes in each generated program. For example, to right-shift a signed integer value `i` by a signed value `n`, our program generator does not output `(i >> n)` but instead outputs `rshift_s_s(i, n)`. The `rshift_s_s` function (*right-shift, signed, signed*) checks the values of its arguments. If the result of `i>>n` is well defined, the function returns that value, and otherwise it simply returns `i`. We implement a family of inlinable wrappers for shifts, division, and modulus.

Allowed implementation-defined behaviors. We have chosen to allow certain constructs in our generated programs whose behavior is not completely specified by the C standard. We make this choice when both (1) avoiding the behavior would unduly constrain our random program generator, and (2) the unspecified behavior does not matter in practice for our testing. For example, C programs commonly use expressions that combine arithmetic and bitwise operators. The representation of integers is implementation-defined in C, so a bitwise operation on the result of an arithmetic expression could technically have differing results across compilers or platforms. In practice, however, all of the platforms in our study use two's-complement integer representations, so we expect that mixing arithmetic and bitwise operators will never lead to varying program behaviors. Moreover, disallowing such expressions would remove a practically interesting set of test programs from our study. We also allow signed integer overflow and underflow—behaviors that are undefined by the C standard—because the GCC-based compilers that we test (Section 6) accept a flag that defines signed overflow as "wrapping around" using two's-complement behavior. We judged that statically guarding against signed overflow was too difficult, and that dynamically guarding against it would overly constrain program behavior.

The rules for integer promotions also introduce implementation-defined behaviors into our test programs. A promotion is a conversion from an integer type to either an `int` or `unsigned int`. These occur at many points in the evaluation of C expressions—e.g., for the operands of shifts—and are not avoidable in our generated programs. Promotions potentially cause implementation-defined behavior because the size of an `int` or `unsigned int` is implementation-defined. The result is that some of our generated tests produce different results across the platforms we use. This is not a problem for our testing, however, because we do not require that the output of a test program be the same across different architectures (Section 6.1).

Dealing with unspecified order of evaluation. Our program generator deals specially with two of the most significant unspecified behaviors in C programs: the order in which the parts of an expression are evaluated, and the order in which the arguments to a function call are evaluated. Of course, programmers deal with this problem by writing C programs carefully, using only expressions and function calls whose results are independent of the order in which their subparts are evaluated. Breaking this coding rule is a well-known source of bugs. Our *randprog* generator enforces this coding style on the programs that it creates.

As *randprog* constructs an expression, it tracks the *read/write effect* of the (partial) expression that has been built. This effect is described by (1) the set of variables that may be read during the evaluation of the expression; (2) the set of variables that may be written during evaluation; and (3) a flag that is set if any of the may-read or may-write variables are volatile. For example, the read-set of `(x || y)` is `x` and `y`, although the read of `y` depends on the dynamic value of `x`. If either `x` or `y` is volatile, then the flag in the read/write effect of the expression will be true, indicating that evaluation will potentially cause a side-effect through a volatile variable access. It is straightforward to construct the effect of an expression by taking the union of the effects of its parts. Similarly, the effect of a function call is the union of (1) all the effects of the actual arguments and (2) the summary effect of the function itself, as described below.

When *randprog* wants to extend an existing expression *expr* by incorporating a new (randomly generated) expression *elem*, it first checks the effect of *elem* for conflicts with the effect of *expr*. A conflict occurs when:

- *elem* may read a variable that is possibly written by *expr*;
- *elem* may write a variable that is possibly read or written by *expr*; or
- *elem* and *expr* each have a volatile access (read or write).

If a conflict is found, the new element is discarded and *randprog* makes a different random choice. The rules for detecting conflicts are loosened when the operator that combines *expr* and *elem* establishes a well-defined evaluation order, as in `(`*`expr`*` || `*`elem`*`)`.

The process of tracking effects ensures that no subpart of an expression can store data to a variable that some other subpart might read or write. Therefore, the value of the expression is independent of the order in which its subparts are evaluated. Our program generator uses the same effect-tracking approach to ensure that the arguments to function calls are independent of evaluation order. Finally, *randprog* tracks the overall effect of every function that it generates. Each function is associated with its visible effect, which is the sets of global variables that may be read and written as a result of a call to the function, and a flag that is set when any of the accessed variables are volatile. Function effects are used, of course, to detect conflicts when a function call is considered for inclusion in some larger expression.

To recap, our effect-tracking system ensures that within our generated test programs, the order of accesses to volatile variables does not depend on unspecified order-of-evaluation behavior. In any context where the order of evaluation is unspecified, *randprog* allows at most one volatile access (read or write). This is a conservative model: in practical software, it may be acceptable for some volatile accesses to occur in an unspecified order relative to each other. Nevertheless, our model is well suited to compiler testing and effective for exposing numerous compiler defects in practice, as we describe next.

4. TESTING FOR VOLATILE CORRECTNESS

In Section 3 we showed how to randomly generate closed, "nearly strictly conforming" C programs that use volatile variables. A bit of additional work is needed to turn these programs into a usable compiler-testing technique.

The technique we developed is called *access summary testing*, which works as follows. For each randomly generated C program (i.e., each test case):

1. Compile the program using the compiler and compiler options that are being tested.
2. Run the compiled program in an instrumented execution environment that logs all memory accesses to global variables.
3. Map accesses in the log to program variables, and filter out all accesses that are not to volatile variables.
4. Create an access summary by applying a summarization function to the log of volatile variable accesses.
5. Compare the observed access summary with the correct access summary for the test case.

The first step—compilation—is generally straightforward, though we found that some of the programs generated by *randprog* would cause one or more of the C compilers in our study to crash. As we describe later in Section 6.1, we exclude these test cases from our experimental results.

Test case execution. Assuming that the test program was compiled successfully, we need to determine if it was compiled *correctly* according to some metric of correctness. Our metrics are based on the output behavior of the compiled program, where the outputs include both a computed data value and a sequence of accesses to volatile variables (Section 3.2). To observe these outputs, we execute the compiled program within an instrumented environment that allows us to capture the program's memory behavior. We developed two such environments that are tailored for access summary testing.

The first is based on Valgrind [15], an open-source binary instrumentation platform. Using Valgrind, we created a tool that adds instrumentation to every memory access within a program. We call this tool *volcheck*, and we use it to monitor IA32 binaries running atop Linux.

The Valgrind platform manages the instrumentation and execution of the test program that is being examined. As new parts of the (binary) program are reached during execution, Valgrind invokes *volcheck* to instrument the binary code fragments. Valgrind's API allows our tool to add instrumentation to the test program without affecting the test program's behavior. At every (static) memory access in the binary, *volcheck* inserts a call to an event-tracing function. After running our tool, Valgrind executes the newly instrumented code, which causes the event-tracing function to be invoked. That function then records the memory access: address, access size (in bytes), and type (read or write). Information about each access is printed to stdout as the accesses occur. This includes the data described above, as well as the name of the accessed variable (if known) and the source program location (if known).

Our second execution environment for access summary testing is based on Avrora [20], a cycle-accurate simulator for wireless sensor networks. It is highly extensible and, in fact, it comes with a "memory monitor": an extension for logging the total number of loads and stores made to each memory location by the program(s) being simulated. Thus, performing access summary testing using Avrora is straightforward. We only needed to write a program to turn Avrora's memory monitor output into access summaries. To terminate a simulation in Avrora, an AVR-specific function in *randprog*'s runtime code places the checksum into a well-known location and then executes the processor's "break" instruction. The break instruction is intercepted by a small Avrora extension that we wrote, which prints the checksum to stdout and then exits.

As described in Section 3.1, the test programs created by *randprog* are not guaranteed to terminate. Therefore, when we execute a test program within our testing procedure, we impose a timeout. If a program runs too long, we abort it and exclude it from our experimental results.

Summarization. From the log produced by either of our execution platforms, we can easily produce a summary of the volatile-variable access patterns of a test program. We inspect the program source code to find the volatile variables, and we inspect the object code to find the variable addresses. Using this information, we filter the execution log to produce the summary we want. The summary is the primary measure that we use to decide if the volatile-access behavior of a test case is right or wrong. (We also check the value of the output checksum, as described in Section 6.)

The summarization function that we use computes the total number of loads from and stores to each volatile location across the entire execution of the program. We believe this to be a good choice because summaries are compact and they are independent of the order of accesses. Recall from Section 2 that a C compiler may reorder volatile accesses that occur between sequence points. Our *randprog* generator avoids producing code that would be subject to such reordering, but the metric is chosen to be general and to accommodate future changes to our test program generator.

It would be possible to use more abstract summarization functions: for example, mixing loads and stores, or mixing accesses to multiple variables. Increasing abstraction may lead to more efficient testing but risks missing bugs. Similarly, more concrete summarization functions could also be used: e.g., splitting out results by function, by basic block, or even by sequence point. Increasing concretization adds complexity to the program instrumentation and may result in large summaries, but potentially catches more bugs.

Determining correct volatile-access behavior. For hand-written test cases, we usually know what the access summary should look like. In contrast, for randomly generated programs, we do not know how many times each volatile variable should be loaded and stored. In practice this is not a problem since we have many compilers to choose from and many combinations of optimization flags for each compiler. We can compute access summaries for many different versions of a given program and then use a voting scheme to identify the summary that is most likely correct.

When we discover an access summary with an incorrect volatile-access pattern, we have found a test case that shows a *volatile error*, which may be due to a *volatile bug* in the compiler under test.

5. WORKING AROUND VOLATILE BUGS

Our hypothesis was that volatile accesses are more likely to be compiled correctly if they are hidden (or partially hidden) behind function-call boundaries. The intuition is that we can replace an action that compilers empirically get wrong by a different action—a function call—that compilers can get right.

To test our hypothesis, we devised a way to rewrite programs so that volatile variables are accessed through type-specific helper functions. For example, suppose that `x` is a volatile integer, `y` is a pointer to volatile integer, and `z` is a volatile pointer to integer. Our program transformation would then rewrite this code:

```
x =  x;
y =  y;
*y = *y;
z =  z;
*z = *z;
```

into this code:

```
      *vol_id_int(&x) =  vol_read_int(&x);
                    y =  y;
       *vol_id_int(y) =  vol_read_int(y);
   *vol_id_intptr(&z) =  vol_read_intptr(&z);
*vol_read_intptr(&z) = *vol_read_intptr(&z);
```

Note that the second line does not require rewriting: a pointer to volatile is not itself volatile.

The bodies of the integer helper functions are:

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

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

The helper for reading from a volatile integer does exactly what one would expect: it dereferences the pointer and returns the value read. However, in order to simplify program transformation, we do not have a helper function for writing to the volatile. Rather, we "trick" the compiler by passing the target address of the write to an identity function that merely returns its argument. Because the helper function is opaque, the compiler treats the returned pointer as fresh and—for all compilers that we tested—reliably dereferences it. The alternative of performing the store-to-volatile in the helper is not difficult, but it is syntactically more intrusive.

Note that it is critical that the helper functions are not inlined. If inlining is performed, the resulting code (after straightforward optimizations) is precisely the same as if the function call had never been introduced in the first place.

We have two implementations of this transformation. First, we modified *randprog* to optionally wrap all accesses to volatiles in the programs that it creates. Second, we implemented an automatic source-to-source transformation using CIL [14] that wraps accesses to volatile variables in arbitrary C programs. In both of our implementations, the transformation and the generated helper functions are nearly identical to the example code shown above.

6. EXPERIMENTAL RESULTS

We now describe our experiments in testing production-quality C compilers for bugs related to accesses of volatile variables. By hand, and using access summary testing, we found defects in all of the thirteen compilers we tested. We present the benefits and costs of working around volatile-access bugs by introducing helper functions. In our experience, the use of helper functions is extremely effective for avoiding volatile errors. We detail three compiler defects that we found using access summary testing. Finally, we present recent improvements in one of the compilers in our study.

6.1 Methodology

We selected thirteen C compilers for our experiments: nine versions of GCC [3] and one version each of LLVM-GCC [12], Intel's C compiler [6], the Sun Studio C compiler [19], and Freescale's CodeWarrior Development Studio C compiler [4]. The versions and target platforms of these compilers are shown in Table 1. CodeWarrior was hosted on Windows XP; all of the other compilers were hosted on Ubuntu Linux 7.04.

We first tested these compilers by hand, using a small set of functions resembling the `reset_watchdog` function shown in Section 1. We inspected the assembly outputs of the compilers by hand to find errors in the handling of volatile variables.

We then tested nine compilers more thoroughly: we generated random test programs via *randprog* and applied access summary testing. We generated and tested C programs until we had 250,000 valid test cases, where a valid test case is one that:

- caused no compiler to crash or run for more than 30 seconds,
- terminated within 15 seconds, and
- accessed at least one volatile variable at least one time (outside the checksum computation).

We classify a valid test case as a *volatile error* for a compiler if the volatile-variable access summary of the program changes when the program is compiled at different levels of optimization. We classify a valid test case as a *functional error* for a compiler if the checksum computed over the program's global variables changes when the program is compiled at different levels of optimization. In other words, a functional error indicates a case in which the compiled program computes the wrong result. A single test case can be both a volatile error and a functional error.

For classifying errors, we did not require that the access summaries or checksums for a test case agree *across* compilers. Rather, we counted a test case as an error for a compiler when the output of the test case varies across that *one* compiler's optimization levels. It would be desirable to check outputs across compilers, but we did not do so for three reasons. First, we have no automated way to decide which compiler is at fault when there is disagreement. Second, across compilers and platforms, some differences are due to implementation-defined behaviors that we cannot reasonably avoid (Section 3.2). Third, in our experience, our intra-compiler metric is sufficient and effective for finding compiler defects.

For each GCC-based compiler (including LLVM-GCC, which uses GCC's front end), we tested five optimization options: `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. The `-Os` option is commonly used when compiling embedded software as it is intended to minimize the size of the generated code. We also passed the `-fwrapv` option to GCC and LLVM-GCC, which instructs these compilers to provide two's-complement semantics when signed integers overflow or underflow. As explained in Section 3.2, this reduces the need for guard functions within the randomly generated test programs.

To study the impact of adding helper functions, we measured the object code sizes of the compiled, randomly generated programs. For each program, we took its code size for a given compiler to be the smallest size across all of the program's compiled versions. (No single optimization level consistently produces the smallest code.)

6.2 Volatile errors

The "volatile errors" column in Table 1 shows that no compiler that we tested was free from defects: no compiler was able to always create executables that produce the same access summary across all optimization options. From this data, one *cannot* conclude that the percentage of generated programs that elicit buggy compiler behavior is correlated with either:

- the number of bugs in that compiler, or
- the likelihood that that compiler will miscompile real embedded applications.

An interesting trend in Table 1 is the apparent increasing bugginess of the sub-versions of GCC version 4 for IA32. Our (entirely

compiler	version	target	volatile errors (%)	volatile errors w/ helpers (%)	vol. errors fixed by helpers (%)	avg. code size increase due to helpers (%)	functional errors (%)
GCC	3.4.6	IA32	1.228	0.300	76	1.5	0.004
GCC	4.0.4	IA32	0.038	0.018	51	2.3	0.031
GCC	4.1.2	IA32	0.195	0.016	92	3.4	0.025
GCC	4.2.4	IA32	0.766	0.002	100	3.5	0.003
GCC	4.3.1	IA32	0.709	0.000	100	4.0	0.003
LLVM-GCC	2.2	IA32	18.720	0.047	100	2.1	0.126
GCC	3.4.3	AVR	1.928	0.434	77	0.2	0.391
GCC	4.1.2	AVR	0.037	0.033	10	0.8	0.254
GCC	4.2.2	AVR	0.727	0.021	97	0.8	0.214
CodeWarrior	6.4.0.6	Coldfire	*volatile access errors verified*				
GCC	3.2.3	MSP430	*volatile access errors verified*				
Intel	10.1.012	IA32	*volatile access errors verified*				
Sun	5.9	IA32	*volatile access errors verified*				

Table 1: Results from applying access summary testing to randomly generated C programs. For the last four compilers listed in the table, we have verified at least one instance of incorrect code generation, but we either lack results from random testing (on architectures other than IA32 and AVR) or cannot provide details due to license restrictions.

unsubstantiated) hypothesis is that because GCC 4.0 represented a major revision of the basic infrastructure (it was the first version to be based on SSA), it was a clean design that was largely capable of compiling accesses to volatile variables. Subsequent sub-versions of GCC 4 implemented increasingly aggressive optimizations, possibly leading to more and more bugs in accessing volatile variables.

6.3 Impact of helper functions

We evaluate the effect of introducing volatile-access helper functions by answering several questions.

Does the introduction of helper functions improve code correctness? Aggregating across all compilers for which we present quantitative results, 96% of all the volatile errors we found are fixed through the introduction of helper functions.

Why are helper functions not always successful in working around volatile bugs? In our experience, when a volatile error is not fixed by our refactoring, the problem is due to functional bug that manifests as a volatile error. A functional bug is a compiler defect that causes a program to compute incorrect data—e.g., control-flow bugs. We discuss these further in Section 6.4. We investigated several cases where helper functions failed to fix a volatile error, and in each case, the root cause was a functional bug.

What is the overhead of introducing helper functions? The "average code size increase" column in Table 1 shows that for our generated programs, the average code size overhead of helper functions was not more than 4% for any compiler that we tested.

To determine the overhead of helper functions for real embedded software, we used CIL to wrap all accesses to volatiles for a collection of applications based on TinyOS 2 [9], a popular software platform for wireless sensor network devices. We evaluate the impact in terms of code size and *duty cycle*—the fraction of time that a node's CPU is active. Duty cycle is a good efficiency metric because the CPU uses much more power when it is active, and low energy usage is a typical requirement for wireless sensor nets.

For very small applications such as "Blink," which uses a timer to flash a node's LEDs, the code-size overhead was around 70% and duty-cycle overhead around 25%. For larger applications such as "MViz" and "MultihopOscilloscope," the code-size overhead was in the 10–15% range but the duty-cycle overhead was 70–80%. We believe these rather high overheads are due to the fact that sensor-network applications are extremely low-level and spend much of their time interacting with memory-mapped peripherals.

6.4 Causes and effects

Compilers have two distinct kinds of code generation bugs that affect our study: volatile bugs and functional bugs. A *volatile bug* may result in an executable that violates the access requirements of volatile variables (i.e., exhibits a volatile error), but that will never cause our generated test programs to be functionally incorrect. A *functional bug* may result in an executable that computes the wrong checksum (i.e., a functional error) and may also change the volatile-access behavior of the generated code (a volatile error), for example by taking the wrong branch of a conditional.

The following table describes the relationship between compiler bugs and observed errors in our randomly generated test programs:

	. . . may result in			
a compiler defect. . .	crash or hang	functional error	volatile error	no symptom
functional bug	✔	✔	✔	✔
volatile bug			✔	✔

An observed functional error implies that a functional bug exists. However, an observed volatile error does not imply that a volatile bug exists: the error may stem from a functional bug. Finding a volatile bug, therefore, requires examining the compiler's source code. In general, we have found it fruitful to investigate test cases that are volatile errors but not functional errors. As the data in Table 1 show, such test cases are generally not difficult to find. In our test suite, for most compilers, the number of volatile errors is much greater than the number of functional errors.

6.5 Example volatile bugs

To illustrate the character of bugs found by access summary testing of randomly generated C programs, we describe in detail two volatile bugs in GCC 4.3.0 and one in LLVM-GCC 2.2. These bugs manifest on IA32, the compilers' most important and most heavily tested target, and do not require exotic compiler flags or code constructs. All three bugs have been reported to the respective compilers' developers, and two of them have been confirmed and fixed. The C functions presented in Figures 1–3 are subsets of randomly generated programs that were flagged as volatile errors by our automated testing framework.

```
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     .L2
.L3:
  movl    y, %eax
  incl    %eax
  movl    %eax, y
.L2:
  movl    y, %eax
  cmpl    $10, %eax
  jg      .L3
  ret
```

Figure 1: At the `-Os` optimization level, GCC 4.3.0 for IA32 compiles the C code at left to the assembly at right. The compiler output is wrong: the access to `volatile`-qualified variable `x` should not be hoisted out of the loop.

```
extern int qux();
volatile int w;
int bar(void) {
  if (qux())
    return 0;
  else
    return w;
}
```

```
bar:
  subl    $12, %esp
  call    qux
  cmpl    $1, %eax
  sbbl    %eax, %eax
  andl    w, %eax
  addl    $12, %esp
  ret
```

Figure 2: At the `-O` optimization level, GCC 4.3.0 for IA32 compiles the C code at left to the assembly at right. The compiler output is wrong: it unconditionally accesses `volatile`-qualified variable `w`.

GCC bug #1. This bug is shown in Figure 1.[3] The source code clearly specifies that `x` should be loaded on each loop iteration. However, GCC incorrectly recognizes `x` as loop-invariant and hoists the load out of the loop. In the object code, `x` is loaded once instead of zero times (or however many times the loop ends up executing, if volatile `y` does not behave as a normal variable).

We reported this bug to the GCC developers, who rapidly fixed it. The problematic code was in a file called `loop-invariant.c`, which contained the following check to determine if it is safe to hoist a load from a constant memory location out of a loop:

```
if (MEM_READONLY_P(x)) ...
```

The fixed code reads as follows:

```
if (MEM_READONLY_P(x) && !MEM_VOLATILE_P(x)) ...
```

GCC bug #2. This bug is shown in Figure 2. The correct behavior of this function with respect to volatile variable `w` is clear: if `bar` returns zero then a single load to `w` should be issued, and otherwise `w` should not be touched. In fact, the code emitted by GCC loads from `w` regardless of the return value of `bar`. The fact that `w` is always loaded is obvious from the lack of branches in this code.

The compiler is being clever in order to create straight-line object code from conditional source code. The `sbbl` (subtract with borrow) instruction is used to subtract `eax` from itself, which clears the register unless the carry bit flag is set, in which case `eax` is set to –1, i.e., all ones. Either way, the subsequent bitwise-and of the value found in `w` with `eax` causes the proper value to be returned from this function. The bug in GCC is that the machine-specific optimization pass that creates the control-flow-free code is insufficiently respectful of `w`'s `volatile` qualification. We reported this bug to the GCC developers, but as of this writing, it has not yet been confirmed or fixed.

[3] To get the assembly code shown in Figures 1–3, we passed the compiler the `-fomit-frame-pointer` flag to make the assembly code easier to read. The bugs are all independent of this flag.

```
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
```

Figure 3: At the `-O2` optimization level, LLVM-GCC 2.2 for IA32 compiles the C code at left to the assembly at right. The compiler output is wrong: it loads from `a` once instead of three times.

An LLVM-GCC bug. Figure 3 illustrates a problem in compiling a C function with a loop. The loop executes three times and increments a volatile location by seven on each loop iteration. The compiler completely unrolls the loop, which is fine, but a subsequent optimization pass improperly caches the incremented value of `a` in a register instead of reloading it.

We reported the bug to the LLVM developers and they rapidly fixed it. As with the GCC bug above, the problem was a missed condition in an optimization safety check. The buggy compiler code contains this test:

```
if (LD->getExtensionType() == ISD::NON_EXTLOAD)
  ...
```

It was fixed by adding a condition:

```
if (LD->getExtensionType() == ISD::NON_EXTLOAD &&
    !LD->isVolatile()) ...
```

6.6 Toward zero volatile bugs

We ran an experiment to see how low we could drive the volatile-error rate of a compiler. We chose LLVM as our target because the developers were very responsive to our reports, fixing most bugs within a few days. Between March and July 2008 the LLVM team fixed five volatile bugs and eight functional bugs we reported. Measured by our 250,000 test cases, the improvements are as follows:

LLVM-GCC for IA32 version	volatile errors (%)	volatile errors w/ helpers (%)	errors fixed by helpers (%)	functional errors (%)
2.2	18.720	0.047	100	0.126
r53339	0.002	0.002	0	0.009

LLVM 2.2 was released on February 11, 2008, and LLVM r53339 is a snapshot of the source code from July 9, 2008. We employ a snapshot rather than a released version of LLVM because, as of this writing, no release incorporates fixes to all of the bugs we reported.

The correctness of LLVM-GCC increased dramatically in just a few months: functional errors were reduced by a factor of 14 and volatile errors by a factor of more than 9,300. Of course, during the interval between the two versions, the LLVM developers fixed many bugs besides the ones that we reported. Although it would have been possible to isolate the changes that were in response to our bug reports by manually backing out selected compiler patches, in practice these patches were sometimes entangled with other changes and we judged that isolating the effects of our bug reports was too difficult. Our experiment is ongoing and we expect that within the foreseeable future, LLVM-GCC will be effectively free of volatile and functional bugs for the programs emitted by *randprog*.

7. DISCUSSION

We now discuss some of the broader implications of our findings.

7.1 Why are compilers so buggy?

Our educated guess is that there are several reasons why problems in compiling volatiles are so widespread. First, `volatile` is in tension with optimizations. Compiler writers are always under pressure to produce better code, and a broad class of optimizations can result in violations of volatile semantics if care is not taken. Second, without a technique such as access summary testing, the compilation of volatiles is hard to test, and hence compiler bugs in translating volatile accesses are not easy to notice. Third, compilers tend to be tested by compiling themselves, benchmark suites, and so on. Desktop applications interact with the OS entirely through the system call interface, which stresses a compiler's handling of function calls but not its handling of volatile accesses.

It is interesting that random testing showed that the nine compilers we tested extensively all contain functional bugs for integer programs. Compilers are large, complex artifacts and must be tested aggressively.

7.2 Why can volatile bugs be overcome through helper functions?

We believe that the answer is simple: the rules for optimizing accesses to volatile variables are quite unlike the rules for optimizing accesses to non-volatile variables, but are similar to the rules for optimizing external function calls. In other words, calls to external functions, like accesses to volatile variables, must not be duplicated, eliminated, or moved around very much. The machinery in a typical compiler for making the right number of function calls, and for making them in the right order, tends to be thoroughly tested—or else the compiler would not work even in a basic way. Associating volatile accesses with helper functions reuses this heavily tested machinery, as opposed to using the (empirically) much weaker logic that protects volatile accesses from optimizations.

7.3 Recommendations for application developers

We recommend that the developers of mission-critical or safety-critical embedded software that is written in C and that relies on `volatile` take one or more of the following steps:

- *Manually validate the compiler's output* for functions that rely importantly on proper compilation of accesses to volatile variables.
- *Develop specific tests* to ensure that the compiler generates proper code for the kinds of volatile accesses found in the software of interest.
- *Factor accesses to volatiles into small helper functions*, if the overhead of this can be tolerated.
- *Compile functions that critically rely on* `volatile` *with optimizations turned off*, if the overhead of this can be tolerated. Our belief (based on observations, but unsupported by systematic data) is that code emitted by compilers for accessing volatiles is generally correct when optimizations are turned off. It is folklore among embedded software researchers that in many cases, safety-critical embedded software for applications such as commercial avionics is always compiled without optimizations.

7.4 Recommendations for compiler developers

Our first recommendation is for C compiler developers to perform access summary testing as part of the compiler's regular test plan. The main requirement for performing this kind of testing is the ability to compute access summaries. This is easy, given the large number of simulators, emulators, and binary translators that are available for common platforms.

Our second recommendation is that compiler developers should look for ways to reuse the dependency infrastructure surrounding function calls in order to avoid miscompiling accesses to volatile storage locations. This would seem to provide a way to avoid the optimization safety checks that were seen in Section 6.5 to be missing in mature, high-quality compilers.

7.5 Recommendations for the C standards committee

First, the C standard states: "What constitutes an access to an object that has volatile-qualified type is implementation-defined." This is a mistake. A strong fix would be to require compilers to generate an error if the minimum memory-access granularity forces an access to one volatile object to improperly access a second volatile object. Statically detecting this condition at link time should be straightforward.

Second, the interaction of `volatile` with weak memory models should be clarified. In particular, if a conforming compiler is required to insert barrier instructions at sequence points in order to force volatile accesses to commit to RAM in the proper order and in a timely fashion, this should be made explicit. Boehm and Maclaren [1] have addressed this and related issues.

7.6 Future work

We would like to develop a static version of access summary testing, perhaps called *access summary analysis*. This would be an abstract interpretation of object code that computes a conservative access summary.

As far as we know, a formal model of the behavior of volatile variables has never been constructed. Adding such a model to an existing semantics for C would be a useful step towards verifying compilations of embedded software.

8. RELATED WORK

Although the correctness of systems software relies critically on it, the `volatile` qualifier has received almost no attention from the academic compiler community.

In fact, only one testing effort that we know of—*NULLSTONE for C* [16]—addresses the correctness of `volatile` at all. The NULLSTONE for C Web site states that in a 1995 study, "eleven out of twelve commercially-available C compilers failed one or more NULLSTONE volatile conformance tests." NULLSTONE employs a fixed test suite, whereas we use random testing. Our experience is that both techniques are useful and effective for finding compiler defects. Moreover, both should be used regularly: our results show that volatile problems are at least as widespread today as NULLSTONE reported them to be in 1995. We are not aware of any previous work on automated, random testing of C compilers' implementations of the `volatile` qualifier.

Embedded C [8] is a proposed set of extensions to C, intended to support embedded systems programming. It defines a suite of I/O hardware access functions, in `<iohw.h>`, that abstract over many common uses of volatile variables in embedded systems. On the surface this seems similar to our wrappers (Section 5). However, the goal of our wrappers is not to provide an architectural abstraction for programmers: it is simply to "trick" existing compilers into producing correct code. Furthermore, the Embedded C draft shows how to implement its `<iohw.h>` interfaces via macros, in a way that results in inlined code. In contrast, for our wrappers to have

the desired effect on volatile correctness, it is important that they not be inlinable by the compiler.

Previous work shows that both random and non-random program generation can be effective for finding defects in C compilers. Lindig [10], for instance, used randomly generated programs to find several compiler bugs related to calling conventions. McKeeman [13] described the benefits of *differential testing* in conjunction with random program generation. Two or more C compilers are compared with each other over a series of random programs: any differences in the behaviors of the compilers or the compiled test programs are indicators of bugs. Sheridan [17] described a similar method, in which multiple compilers are compared over non-randomly generated inputs. Both McKeeman and Sheridan report that their approaches uncovered numerous bugs in production-quality compilers. The testing methodology we use is most similar to McKeeman's, but our work is targeted toward a specific class of bugs (similar to Lindig's).

There are a number of random C program generators (e.g., [11, 21]) and C compiler testing frameworks (e.g., [2]) available on the Web. To our knowledge, however, only ours is tailored for finding defects in the handling of volatile objects.

9. CONCLUSION

A code generation bug in a compiler used for embedded software and operating systems represents a very serious problem, potentially placing many mission- and safety-critical systems on an unsound foundation. This paper has shown that the foundation is indeed not solid: all thirteen production compilers that we tested had problems in translating C code for accessing `volatile`-qualified objects. Low-level C code relies on the `volatile` qualifier to access memory-mapped I/O devices and to implement communication between concurrent threads and interrupts.

Our main contribution is a method for automatically detecting bugs in compiling accesses to volatile objects. First, we developed a way to randomly generate closed and "nearly strictly conforming" C programs that are useful and effective test cases for C compilers. The important property of these programs is that they must—with a few exceptions as suggested by "nearly"—perform the same sequence of accesses to volatile objects, regardless of compiler, target platform, or optimization level. Second, we developed access summary testing, which detects violations of these programs' required volatile-access patterns. A secondary contribution of our paper is the result that, over the randomly generated programs in our study, 96% of the failing test programs could be made to succeed (i.e., be correctly compiled) by inserting small helper functions into the test program source. Based on our findings, we developed a collection of recommendations for application developers, for compiler developers, and for the C standards committee.

Software. The software that we developed for this work is available at `http://www.cs.utah.edu/~eeide/emsoft08/`.

10. ACKNOWLEDGMENTS

We thank Michael Norrish, Alastair Reid, Jonathan Shapiro, and Eric Weddington for providing helpful feedback on drafts of this paper; Nathan Cooprider for implementing the CIL transformation that refactors accesses to volatiles; and the LLVM team for rapid and friendly responses to our bug reports. To generate and evaluate the 250,000 C programs described in Section 6.1, we used a cluster of machines within Emulab, a testbed for network and systems research [22]. This material is based upon work supported by the National Science Foundation under Grant No. 0448047.

11. REFERENCES

[1] Hans Boehm and Nick Maclaren. Should volatile acquire atomicity and thread visibility semantics? WG21 Paper N2016, ISO, April 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html`.

[2] Digital Equipment Corporation. DDT kits. `http://nar-associates.com/site/sdr/projects/ddt/download/kits/`, December 1996.

[3] Free Software Foundation. GCC, the GNU compiler collection. `http://gcc.gnu.org/`, 2008.

[4] Freescale Semiconductor, Inc. CodeWarrior Development Studio. `http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=0127262E70`, 2008.

[5] Douglas Gwyn. NEVERMIND! (was: Re: A question on volatile accesses). USENET post to comp.std.c, November 1990. `http://groups.google.com/group/comp.std.c/msg/7709e4162620f2cd`.

[6] Intel Corporation. Intel C++ Compiler 10.1, professional and standard editions, for Linux. `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/277618.htm`, 2008.

[7] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming Languages—C*, May 2005. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[8] International Organization for Standardization. Information technology — programming languages – C – extensions to support embedded processors. ISO/IEC draft Technical Report 18037, ISO, April 2006. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf`.

[9] Philip Levis et al. T2: A second generation OS for embedded sensor networks. Technical Report TKN–05–007, Telecommunication Networks Group, Technische Universität Berlin, November 2005.

[10] Christian Lindig. Random testing of C calling conventions. In *Proc. 6th Intl. Symp. on Automated Analysis-Driven Debugging (AADEBUG)*, pages 3–12, September 2005.

[11] Christian Lindig. Quest – an extensible test-code generator for C. `http://code.google.com/p/quest-tester/`, 2007.

[12] LLVM Team, University of Illinois at Urbana-Champaign. The LLVM compiler infrastructure project. `http://llvm.org/`, 2008.

[13] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, December 1998.

[14] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. Intl. Conf. on Compiler Construction (CC)*, pages 213–228, April 2002.

[15] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 89–100, June 2007.

[16] NULLSTONE Corporation. NULLSTONE for C. `http://www.nullstone.com/htmls/ns-c.htm`, 2007.

[17] Flash Sheridan. Practical testing of a C99 compiler using output comparison. *Software—Practice and Experience*, 37(14):1475–1488, November 2007.

[18] Steve Summit. comp.lang.c frequently asked questions. `http://c-faq.com/`.

[19] Sun Microsystems, Inc. Sun Studio 12 C, C++, and Fortran compilers and tools. `http://developers.sun.com/sunstudio/`, 2008.

[20] Ben L. Titzer and Jens Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proc. 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 59–68, June 2005.

[21] Bryan Turner. Random C program generator. `http://brturn.googlepages.com/randomcprogramgenerator`, January 2007.

[22] Brian White et al. An integrated experimental environment for distributed systems and networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, December 2002.