Adding Support for MC/DC Instrumentation in the Green Hills C/C++ compiler

by

Sagnik Saha

S.B., Massachusetts Institute of Technology (2018)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

Author . . . .
Department of Electrical Engineering and Computer Science
February 1, 2019

Certified by . . . .
Martin Rinard
Professor
Thesis Supervisor

Certified by . . . .
Nikola Valerjev
Director of Engineering at Green Hills Software
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Adding Support for MC/DC Instrumentation in the Green Hills C/C++ compiler

by

Sagnik Saha

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents the design and implementation of Modified Condition / Decision Coverage (MC/DC) instrumentation in the Green Hills C/C++ compiler. When a specific option is enabled, the compiler now identifies each boolean expression and annotates the generated binary with special instructions. When a test suite is run, these extra instructions emit logging information. A separate program then uses that information to determine and display the degree of coverage achieved. Taken together, my tools allow a user to run any program and determine the extent of MC/DC coverage achieved by their tests.

Thesis Supervisor: Martin Rinard
Title: Professor

Thesis Supervisor: Nikola Valerjev
Title: Director of Engineering at Green Hills Software

Contents

1 Introduction **9**

1.1 Overview and Applicability 10

1.2 Standard Coverage Metrics 11

1.2.1 Function Coverage 11

1.2.2 Statement Coverage 11

1.2.3 Branch Coverage 11

1.2.4 Condition Coverage 12

1.2.5 Condition / Decision Coverage (C/DC) 12

1.2.6 Multiple Condition Coverage (MCC) 12

1.3 Modified Condition / Decision Coverage (MC/DC) 13

1.3.1 Definition 13

1.3.2 Example 14

1.3.3 Motivation 15

1.4 Thesis Organization 15

2 Front-End Modifications **17**

2.1 Maintaining Position Information 17

2.2 Handling Inline Functions 18

2.3 Identifying Expressions to Instrument 19

3 Instrumentation Process **21**

3.1 Known Algorithms 21

3.1.1 Naive Implementation 21

3.1.2 Implied Decision Algorithm 22

3.1.3 Masking Algorithm 23

3.2 Preliminary Design 24

3.3 Optimizing single-variable Decisions 24

3.4 Removing the branch instruction from DC instrumentation 25

3.5 Future work 26

3.5.1 Optimizing two-condition decisions 26

3.5.2 Loop Optimization 27

3.5.3 Removing instrumentation when possible 28

4 Supporting utilities **31**

4.1 Interpreting results of instrumentation 31

4.2 Testing 33

5 Experimental Results **35**

5.1 Benchmark Suites 35

5.2 Time Overhead 37

5.3 Space Overhead 40

6 Related Work **43**

7 Conclusion **45**

A EEMBC benchmarks **47**

List of Tables

5.1 Coverage Information of Stanford and Coremark Test Suites 38
5.2 Reduction of Time Overheads due to Single Condition Decision Optimizations 39
5.3 Reduction of Time Overheads after Optimizing Two-condition Decisions and Loops 40
5.4 Reduction of Size Overheads due to Single Condition Decision Optimizations 41
5.5 Reduction of Size Overheads after Optimizing Two-condition Decisions and Loops 41

A.1 Coverage Information of EEMBC test suite 48

Chapter 1

Introduction

Testing code is an integral part of the software development process [17]. It is standard practice to develop a test suite which any new code must pass before being released for production. The quality of industrial software produced often depends heavily on how extensive the test suite is, since developers rely on the tests catching their bugs before the code is released. It is therefore very useful to be able to gauge the effectiveness of a test suite.

Ideally, a test suite would test that the program behaves as expected for every possible input; that would *guarantee* that the program is correct. However, the space of all possible inputs is practically infinite for most industrial programs. Hence, this approach is almost always impossible (except perhaps a few low-level hardware implementations where the number of possible inputs is tractable).

This leads us to the concept of code coverage. Even though it might be impractical to run our code on every possible input, we would like to exercise every part of the codebase as often as possible through the test suite. Intuitively, if running our tests exercises a certain part of our code, we gain confidence on that part of the code if the tests pass. Thus, we can judge a test suite by the parts of the source code it exercises.

1.1 Overview and Applicability

In this thesis, I present a system that helps programmers with the aforementioned evaluation. Using my tools, one can run a test suite on a program and quickly determine the extent of coverage achieved by that suite. Specifically, my system outputs whether MC/DC (defined below in section 1.3) has been achieved by the test suite for each boolean expression in the target program. Note that for many of the metrics mentioned below, it is often impossible to achieve 100% coverage because of the code structure (programmers often build redundancies in their code to improve readability, robustness or even performance). Keeping this in mind, I produce detailed information when maximum coverage is not achieved within a file or a module.

Certain regulatory and certification authorities require full MC/DC coverage. For example, the DO-178B standard requires it for all level A software used in the aviation industry, and automotive standard ISO 26262 highly recommends it for ASIL (Automotive Safety Integrity Level) D software [15]. Software developers in these industries can use tools like mine to determine which parts of their system are insufficiently tested, and can add more tests to achieve the desired level of coverage. In cases where getting 100% coverage is impossible, they have to provide adequate justification in order to get their code certified.

Even when 100% coverage is not an explicit requirement, developers might try to achieve the same anyway to gain confidence in the correctness of their software. For example, SQLite, a very popular database management system, is advertised to achieve full MC/DC coverage with their testing ([10]) and it has a reputation for being highly reliable. In some applications, certain core components are deemed more important to get working correctly, and are subjected to more intensive testing (e.g. OS kernels). Systems like mine can be used to ensure full MC/DC coverage for those components. Lastly, my system can also be used as a profiling tool to just get an idea of how extensively each part of a codebase is tested, and developers can make informed decisions about which components they want to test more.

1.2 Standard Coverage Metrics

The notion of *part* was deliberately left vague earlier. Different definitions of part lead to different coverage metrics. We explore a few of the popular definitions in the subsections below.

1.2.1 Function Coverage

The simplest notion of part corresponds to functions in a program. A test suite satisfies function coverage if and only if all of the functions in the source code are executed when we run our tests. This is one of the weakest metrics in existence.

1.2.2 Statement Coverage

Instead of the functions, we can also consider the statements of the program as its units. Statement coverage monitors whether each statement is executed by the test suite. Note that this is a strictly stronger metric than function coverage. Statement coverage almost always implies function coverage since most functions contain at least one statement. However the reverse is not true, since it's possible to have conditional code within a function that is not executed even though the function itself gets executed.

1.2.3 Branch Coverage

Branch coverage, also known as decision coverage, asks whether every branching point in the program assumed both *true* and *false* values. This is strictly stronger than statement coverage. As long as both branch directions are taken at every decision point, we are bound to execute every statement reachable from the start. But even when statement coverage is achieved, some branches may not evaluate to both values; in C, a **do - while** loop that executes only once provides a simple example.

1.2.4 Condition Coverage

Condition coverage is very similar to decision coverage. Instead of requiring every boolean decision to assume both possible values, this metric requires every individual condition appearing in boolean expressions to take both values. As an example, if we have the following code :

```
if (x && y) {
        ...
}
```

branch coverage requires that the decision in the **if** statement evaluates to both *true* and *false* while condition coverage requires that x and y individually assume both values.

1.2.5 Condition / Decision Coverage (C/DC)

A test suite is said to satisfy C/DC if and only if it satisfies branch coverage as well as condition coverage. Obviously, this is a strictly stronger metric than both condition coverage and decision coverage by themselves.

1.2.6 Multiple Condition Coverage (MCC)

This is one of the strongest coverage metrics in use. MCC requires that every possible combination of boolean assignments to conditions must be covered for each expressions. In the above example, we would require the following 4 assignments:

- $x = true, y = true$
- $x = true, y = false$
- $x = false, y = true$
- $x = false, y = false$

Clearly, MCC is stronger than C/DC. However, for a boolean expression with n conditions, this metric requires 2^n executions with distinct truth value combinations. This exponential blowup makes MCC infeasible in a lot of practical situations.

1.3 Modified Condition / Decision Coverage (MC/DC)

The MC/DC metric was introduced by avionics software development guidance document DO-178B [3]. This document mandated that the test suite for the most critical software (whose failure can lead to loss of life) must satisfy MC/DC, a more stringent form of condition/decision coverage. This metric is weaker than MCC, but is almost as good at catching bugs.

1.3.1 Definition

The following 4 conditions need to be satisfied for this metric :

- Each entry and exit point is invoked.
- Each decision takes every possible outcome.
- Each condition in a decision takes every possible outcome.
- Each condition in a decision is shown to independently affect the outcome of the decision.

The first condition is only slightly stronger than function coverage, and follows from statement coverage. The next two conditions are exactly decision coverage and condition coverage, respectively. The last condition is the main feature distinguishing MC/DC.

A condition independently affects the outcome of the decision if and only if flipping the value of that condition keeping the other values constant *might* change the decision outcome.

1.3.2 Example

Consider the boolean expression

```
d = a && (b || c)
```

The full truth table corresponding to this is given below:

Row	a	b	c	d
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

To satisfy decision coverage, we just need to choose one of the first 3 rows and one of the last 5 rows. To satisfy condition/decision coverage, we could choose rows 1, 6 and 7. However, we need to choose at least 4 rows to satisfy MC/DC.

One combination of rows satisfying MC/DC is rows 2, 3, 4 and 5. Only changing the value of b changes the decision between rows 2 and 4. Similarly, only changing the value of c changes the decision between rows 3 and 4. We haven't chosen such a pair corresponding to a. However, a independently impacts the value of d in row 5 while being false (since flipping a would lead to row 1, where d is true), and it independently impacts the value of d in rows 2 and 3 while being true (since flipping a would lead to rows 6 and 7 respectively, where d is false). Thus we have shown that all the conditions independently impact the decision while assuming both truth values. It is easy to check that condition/decision coverage is also satisfied by these group of rows.

1.3.3 Motivation

Observe that the definition of MC/DC requires 2 separate test vectors corresponding to each condition in a boolean expression; one where the condition evaluates to true and impacts the final decision, and another one where the condition evaluates to false and impacts the final decision. The number of tests needed to satisfy MC/DC is therefore linear in the number of conditions an expression has, which is much smaller than the exponential number required to satisfy MCC. However, the description of the metric is significantly more complicated. Intuitively, MC/DC seeks to guarantee that if your program calculates the wrong value for any one of the conditions, it would calculate the wrong value for the whole decision at least once (corresponding to the test where that condition independently affects the decision).

1.4 Thesis Organization

This project involves significant changes in multiple layers of the compiler. Chapter 2 describes how we modify the front-end to annotate certain nodes in the AST as well as determine which nodes should be processed for instrumentation. Chapter 3 describes the algorithms used to instrument nodes corresponding to boolean expressions and explores various optimizations to improve space and time overheads. In Chapter 4, we discuss the various changes made outside the compiler which were necessary to support MC/DC as a stand-alone feature of the Green Hills developer suite. In Chapter 5, we demonstrate the results obtained using the optimization techniques presented in the thesis.

Chapter 2

Front-End Modifications

The front-end of the compiler is responsible for parsing the source program, performing syntactic & semantic analysis, and producing an abstract syntax tree. In the following sections, I describe the different modifications made to this layer.

2.1 Maintaining Position Information

Like most of the other code coverage metrics we discussed, the definition of MC/DC is very closely tied to the source files of a program. The developer writing the tests needs to know which expressions in the source file were properly covered, and which were not. In order to display this information, we have to keep track of the start and end position (including line number, column and file name) of each expression we instrument.

I added an extra field in the expression nodes used in the AST to hold the position range corresponding to that expression. This field first gets populated as we scan the file to create individual tokens for each symbol appearing in the source code. I also ensure that as we build the tree, each node maintains its position range by properly combining the position ranges of its children.

The front-end often modifies expressions from their original form in the source. For example, during semantic analysis, integer conditions inside an **if** or **while** clause get a $! = 0$ appended to them. It is important to identify each such case and ensure that expression modified by the compiler have the same position range as the original expression they are derived from.

I do not maintain meaningful position information for expression inside macro definitions. This is a deliberate choice. Macro substitution takes place at the preprocessing stage, and the compiler front-end only sees the expanded form. As a result, all such expressions have their position range set to the position range of the macro invocation site. I could have modified the preprocessor to avoid this issue, but that would mean treating each macro invocation as a call to the same body of code, which violates the spirit of macros.

2.2 Handling Inline Functions

For the purpose of code coverage, inline functions should not be treated any differently from normal functions. One reason for this is that the **inline** keyword only serves as a hint to the compiler; the compiler can inline functions which are not so marked (e.g. if it is a member function containing just a simple return expression) and can refuse to inline functions which are marked **inline** without generating an error message (e.g. if the function is too large).

Consider the following code snippet:

```
inline bool foo() {
    return x || y;
}
...
d = foo() && z;
```

There are two boolean expressions in the above code, and they should be instrumented separately. However, the front-end simply replaces the node corresponding to an inline function call by the node corresponding to the appropriate function body, creating one composite expression

```
d = (x || y) && z;
```

Although the above two expressions for **d** are semantically equivalent, MC/DC for the two code snippets entail different requirements and hence, this substitution is illegal when performing MC/DC instrumentation. To prevent this, I create a temporary variable **t** and perform the following replacement:

```
d = (t = x || y, t) && z;
```

Note that this retains semantic equivalence, while not merging the two expressions from the source file. As mentioned in the previous section, this ensures that the position range of the comma expression node I generated is the same as the position range of the function call node in the source (**foo()** in our example).

2.3 Identifying Expressions to Instrument

MC/DC is applicable to all boolean expressions. However, a lot of intuitive approaches to identify such expressions are not general enough.

It is tempting to look at the *type* of an expression to determine whether it is boolean. The problem with this approach is versions of C predating C99 didn't have a **bool** type. Since boolean variables are internally represented by integers, we can't rely on the type of an expression to determine whether the expression should be instrumented or not. Another sensible approach would be instrumenting expressions inside **if**, **while**, **do-while** and **for** clauses (this is a standard approach for implementing branch coverage instrumentation). Although this takes care of most boolean expressions in practice, MC/DC requires even assignments and return statements to be instrumented if they involve boolean expressions. Instead, I use the topmost operator in the AST node corresponding to an expression. An expression is considered boolean if and only if it is a result of a relational ($<, >, <=, >=, ==, !=$) or a logical ($!, ||, \&\&$) operator.

Since the definition of MC/DC mentions individual conditions as well as the final decision, MC/DC instrumentation can't be implemented in a modular fashion. I determine which expressions qualify as *decisions*, and pass only those to our instrumentation function. In order to do this, I added an extra parameter `is_outermost_expression` to the function that processes expression nodes. This parameter is always set to **true** for non-recursive calls and almost always **false** for recursive calls, which is expected. However, there are a few cases where this is true in a recursive call. The first operand of a comma operation is treated as its own expression, not a sub-expression, and is instrumented accordingly. The same holds for each operand of a ternary if, array subscripts, function call arguments and the right-hand side of an assignment operation.

Chapter 3

Instrumentation Process

After the front-end passes certain expression objects for instrumentation, I replace them by functionally equivalent expressions which stores some extra information used for coverage in special sections of memory. This involves the most complicated piece of code in this project.

3.1 Known Algorithms

I experimented with different algorithms for instrumentation. I briefly describe 3 of those algorithms below. I also show the effect of transforming an expression using the algorithm on the following boolean expression :

D = A && (B || C)

3.1.1 Naive Implementation

The simplest algorithm for instrumentation is just recording the value of each condition as well as the decision every time the code is run. If the expression has n conditions, this method requires 2^{n+1} bits to instrument it. Since I use a 32-bit logging variable, this algorithm can handle expressions with up to 4 conditions.

I create an accumulator variable whose first bit represents the decision and the further bits represent the individual conditions in order. As we evaluate each condition, I keep updating this variable. Finally, I use the accumulator variable to get the index of the bit to mark in the logging variable.

The sample expression is turned into the following:

```
D = (_accum = 0,
     _res = ((_temp = A, _accum |= (_temp ? 2 : 0), _temp) &&
             ((_temp = B, _accum |= (_temp ? 4 : 0), _temp) ||
              (_temp = C, _accum |= (_temp ? 8 : 0), _temp))),
     _accum   |= (_res ? 1 : 0),
     _logging |= 1 << _accum,
     _res)
```

3.1.2 Implied Decision Algorithm

We can improve on the previous algorithm by noticing that if we record the value of each condition, the decision can be calculated in the post-processing stage. Thus, we can simply omit recording the decision. If the expression has n conditions, this method requires 2^n bits to instrument it. Since I use a 32-bit logging variable, this algorithm can handle expressions with up to 5 conditions.

The details of this algorithm are almost exactly the same as the previous one, except that we don't need a bit in the accumulator variable for the result. However, I now need to record the expression in order to be able to recalculate the decision.

The sample expression is turned into the following:

```
D = (_accum = 0,
     _res = ((_temp = A, _accum |= (_temp ? 1 : 0), _temp) &&
             ((_temp = B, _accum |= (_temp ? 2 : 0), _temp) ||
              (_temp = C, _accum |= (_temp ? 4 : 0), _temp))),
     _logging |= 1 << _accum,
     _res)
```

3.1.3 Masking Algorithm

The masking algorithm is a space-efficient algorithm for MC/DC instrumentation. This algorithm requires only 2 bits per condition to instrument a boolean expression. Since I use a 32-bit logging variable, this algorithm can handle expressions with up to 15 conditions (2 bits are reserved for the decision). However, the extra complexity needed to save space makes this slower than the previous approaches.

In this algorithm, the i^{th} condition is represented by bits $2i + 1$ and $2i + 2$ of the logging variable. While evaluating the expression, our instrumentation calculates which conditions have an independent effect on the decision with what value in the accumulator variable. The algorithm traverses the boolean expression tree by doing a Depth-First Search. Whenever a leaf node is evaluated, we know that it has an independent effect on the expression at that point. So, if the leaf node had position i in the expression, I set bit $2i+1$ (if the variable was true) or bit $2i+2$ (if the variable was false) in the accumulator. Finally, if the decision overall is false I set the last bit and otherwise, I set the penultimate bit.

However, the result of the evaluation sometimes masks previous subexpressions. If this variable caused the right child of an AND node to be false, the left child of that AND node ceases to independently affect the final expression (because the AND would evaluate to false regardless of its left child). Similarly, if this variable caused the right child of an OR node to be true, the left child of that OR node gets masked out. I keep track of these cases and remove all bits set in the left children getting masked out when applicable.

In some cases, a particular evaluation of a condition determines the final decision. The algorithm also keeps track of when that's the case. In those cases, I flush the accumulator to the logging variable.

This is what our sample expression is converted into:

```
D = (_accum = 0,
      _res =
        (A  ? (_accum |= 4, 1)
            : (_accum |= 8,  _logging |= _accum, 0)) &&
        ((B ? (_accum |= 16, _logging |= _accum, 1)
            : (_accum |= 32, 0)) ||
         (C ? (_accum |= 64,  _accum &= ~48, _logging |= _accum, 1)
            : (_accum |= 128, _accum &= ~12, _logging |= _accum, 0))),
      _logging |= (1 << _res),
      _res)
```

Note that a variable which isn't evaluated due to short circuiting doesn't affect the decision. Bits corresponding to such variables remain 0, which is the correct value. Therefore we don't need any special processing for short- circuited variables.

3.2 Preliminary Design

It's pretty clear from the above discussion that the implied decision algorithm is strictly better than the naive implementation. Therefore, I use the implied decision algorithm for all decisions with less than six conditions. For decisions with 6-15 conditions, I use the masking algorithm. If we encounter a decision with 16 or more conditions, I do not instrument it. A recursive function that traverses the expression tree is used at the beginning to count the number of conditions.

Without any further optimizations, this incurs unacceptably high space and time overheads.

3.3 Optimizing single-variable Decisions

In practice, it turns out that a large fraction of boolean expressions contain only one condition. Most notably, the loop condition for a **for** loop is generally a single comparison and **if** and **while** clauses generally contain a single variable or condition.

Therefore, optimizing our instrumentation for single-condition decisions can lead to significant improvements in overall performance.

Note that the implied decision algorithm transforms a boolean variable `x` into

```
_accum = 0,
_res = (_temp = x, _accum |= (_temp ? 1 : 0), _temp),
_logging |= (1 << _accum),
_res
```

It is easy to see that we do not need so many instructions in this simple case. In fact, all we need to do is to record the value of `x`. Keeping that in mind, we can simplify the above to decision coverage instrumentation as follows :

```
_res = x,
_logging |= (_res ? 2 : 1),
_res
```

Note that although this captures the same information, the format is slightly different. In this version, the last bit records if the *decision* was ever false, whereas in the previous version the last bit records whether the *condition* was ever wrong. Therefore, if the single condition is negated, we should interpret the logging variable a bit differently.

We can use this algorithm whenever the expression to be instrumented contains a single condition.

3.4 Removing the branch instruction from DC instrumentation

Profiling the code shows that majority of the time overhead still comes from instrumenting single-condition expressions. In particular, the branch expression generated by ternary if-expression is expensive. To try to get rid of that branch, it is tempting to rewrite the DC instrumentation code as

```
_res = x,
_logging |= (1 << _res),
_res
```

Since `_res` is a boolean variable the above transformation is almost always valid. However, since boolean variables are internally just integers, the above code doesn't behave as expected if the result variable is greater than 1 (which can happen if we had an integer constant inside an **if** clause, for example).

We can fix the above issue by ensuring that `_res` can only be 0 or 1. A simple way to do that is the following:

```
_res = x,
_logging |= (1 << (_res != 0)),
_res
```

Although this form looks slightly more complicated than the code in the previous section, getting rid of the branch instruction actually reduces timing overhead by a significant margin.

3.5 Future work

The two optimizations mentioned above brought the overhead down to a reasonable level. I also considered a few other optimization ideas which I ended up discarding. Further research into these avenues might produce promising results.

3.5.1 Optimizing two-condition decisions

We saw in section 3.3 that we can use the specific form of the boolean expression to create customized instrumentation code which incurs less overhead than the fully general implied decision algorithm. This insight can be used to optimize boolean expressions with 2 conditions too.

As an example, consider the expression `x && y`. When instrumented, this gets turned into

```
_accum = 0,
_res = ((_temp = x, _accum |= (_temp ? 1 : 0), _temp) &&
        (_temp = y, _accum |= (_temp ? 2 : 0), _temp)),
_logging |= 1 << _accum,
_res
```

We can actually eliminate the accumulator and the temporary variable altogether by replacing the above with

```
x ? (y ? (_logging |= 8, 1) : (_logging |= 4, 0))
  : (_logging |= 1, 0)
```

This expression has the same number of branch instructions, and gets rid of a few assignments and a shift instruction. In fact, we can use a similar trick as in 3.4 to get rid of the branch instruction generated by the inner if-expression if we use the following semantically equivalent code snippet.

```
x ? (_temp = y, _logging |= (4 << (_temp != 0)), _temp)
  : (_logging |= 1, 0)
```

Some special handling would be required if any of these conditions were negated, but the above represents a viable proof of concept. Similar optimizations could be done for `x || y` too.

I decided that the marginal benefit provided by these optimizations is not worth the extra complexity to handle all the cases indicated above at the current stage of development. As these instrumentation tools see more use, I might consider revisiting that decision. Some of these optimizations could also be combined with the loop optimization discussed next.

3.5.2 Loop Optimization

A lot of boolean expressions come from loop constructs. Since these expressions are evaluated quite a lot of times and each evaluation results in at least 1 write to memory (since the logging variable is in memory), it is no surprise that runtime overhead drops significantly once we stop instrumenting loop conditions. Therefore, optimizing instrumentation of these loop conditions could potentially lead to large improvements.

Note that the body of a **for** or **while** loop is executed if and only if the loop condition evaluates to true. Furthermore, the loop condition evaluates to false if and only if we exit the loop *normally* (not through a **break**, **throw**, **goto** or **return** statement). These observations can be leveraged to avoid a linear number of writes to memory in the case where the loop expression has a single condition.

I create a local boolean variable which is set to false at loop initialization. I set it to true at the start of the body of the loop. This variable denotes whether the condition ever evaluated to true. Before every statement inside the loop body that takes the control flow outside the loop (such as **break**), I insert an instruction to set the logging variable to 2 (since the loop condition was set to true but not false). Finally, I create a new basic block between the block for evaluating the loop condition and the first block following the loop such that if the condition evaluates to false, control reaches this new basic block before going to the block immediately after the loop as usual. In this block, I set the logging variable to twice the local variable I created plus 1 (the least significant bit is set since the loop condition was set to false, and the second least significant bit should be set if and only if the loop body was ever executed).

Based on our initial experiments so far, this seems to be a very promising optimization. However, identifying all commands which take control flow outside the loop, and in particular dealing with **goto** statements and nested loops, is quite challenging. Once the feature is somewhat more stable, I plan to implement this optimization fully. It might also be possible to extend this optimization by moving instrumentation instructions out of multiply nested loops.

3.5.3 Removing instrumentation when possible

One interesting feature all the algorithms presented above share is that the bits of the logging variable are never erased. If we somehow knew that a certain bit is already set, we could remove the part of instrumentation responsible for setting that bit.

In practice, a binary is first generated from the source code. We then run the binary against a multitude of tests. Finally, I use logical or to combine the corresponding logging variables from each run and then analyze it. Instead, I could analyze the logging variables after each test is run. If MC/DC is already satisfied for a certain expression, I can remove all the instructions responsible for instrumenting that decision from the binary. In certain cases, I could use an even more precise method to eliminate more instructions (e.g. if a certain condition is covered when using the masking algorithm, I can delete just the instructions instrumenting that variable). This could potentially trim down the binary as we go through our test suite, and both the time and space overhead could go down a lot for the later tests.

It is theoretically possible to remove the extra instructions even earlier if the program has permission to write to the text section. Self-modifying code could get rid of the above instructions as soon as they become redundant, as evidenced in [8], [22] and [6]. If the instrumentation overhead turns out to be too high for users, this could be a very promising avenue of research.

Chapter 4

Supporting utilities

I had to make a few other changes in various parts of the Green Hills code-base to fully support MC/DC instrumentation. The logging variables mentioned above are emitted in a special section; I had to add an option to the linker which includes that section into the section map. The driver (the program that reads input from the command line and calls the compiler and linker with appropriate arguments) also needed a little modification. It now recognizes the option **-mcdc** and passes the aforementioned option to the linker and the debug flag corresponding to MC/DC instrument to the compiler when this option is passed.

4.1 Interpreting results of instrumentation

All the modifications discussed above change the executable file produced when we build a program. Running this executable has all the same effects as running an executable built without my changes, except that it also populates the logging variables with appropriate coverage information. These logging variables are stored in two special sections of the executable file. A separate program was written to interpret these sections and display coverage information in a human-readable way.

Depending on the structure of the expression, I use one of 3 different algorithms I discussed earlier to record information. If the expression has a single boolean condition, I use DC instrumentation as described in section 3.4. If it has between two and five conditions, I use the implied decision algorithm from section 3.1.2. I use the masking algorithm from section 3.1.3 if the expression has between six and fifteen conditions. It is necessary for the interpreter program to know which logging variable represents which expression and which algorithm was used to encode the information present in it. Furthermore, note that the implied decision algorithm relies upon the interpreter to figure out the value of the decision based on the values of its conditions; I therefore need to also record the structure of the expression.

In the compiler, I actually collect all of this information. When an expression is instrumented, I record the name of the file it belongs to as well as its starting and ending line and column numbers. I then traverse the AST corresponding to the expression and record the boolean structure in reverse Polish notation (for example, the expression `a && (b || !c)` is recorded as `A(X)(O(X)(N(X)))`. While traversing, I also record the start and end positions of each of the conditions.

The interpreter program first reads the auxiliary information and counts the number of conditions (`X`s in our notation) to figure out the algorithm used. It then displays the form of the expression and the relevant source position information about the entire decision as well as all the conditions. The rest depends on the specific algorithm used.

If DC instrumentation was used, I only need to figure out if the condition was negated by checking if there is an odd number of `N`s. The last two bits of the logging variable describe if the decision was ever set to false and true, respectively. I use them to determine the values taken by the decision as well as those taken by the condition. Then I display the values taken by the condition, draw the full truth table and indicate which rows were encountered.

If the masking algorithm was used, I don't have enough information to deduce which rows of the full truth table were encountered. Therefore, I simply read the last two bits of the logging variable to determine the values taken by the decision, and keep reading pairs of bits from the right starting at the third least significant bit to determine if each condition impacted the decision while assuming true and false values.

In case the implied decision algorithm was used, I use the form of the expression recorded to build a simplified AST. If the i^{th} least significant bit is set, the indices of the 1 bits in the binary expansion of $i-1$ correspond to the conditions that evaluated to true in one particular execution of the expression. By mimicking the evaluation process on our simplified AST, I can also figure out which conditions were evaluated to false and which conditions were never evaluated due to short-circuiting. Thus, I can record which row of the truth table this execution corresponds to, as well as which conditions impacted the decision with what values. I combine all of this information as I iterate through all the set bits of the logging variable, and then display it.

Sometimes, the same expression has multiple logging variables associated with it. For example, every invocation of an inline function creates a separate logging variable. To handle these correctly, our interpreter program doesn't display the coverage information as soon as a logging variable is processed. Instead, it stores all the relevant information in a giant class. After all the variables are processed, it identifies entries corresponding to the same expression (the source information must match exactly) and merges them. Once all the entries in our class have distinct source positions, I group them by file name, and display them in order of line number.

4.2 Testing

The testing of this feature was mostly done manually. Our test suite consists of functions with different kinds of boolean expressions in them. I call these functions with different combinations of arguments, and check if the coverage information displayed for each function is correct.

I've tested every possible combination of arguments for all possible boolean expressions with one and two conditions. I also test all possible boolean expressions with three conditions; if the truth table has four rows I try all combinations of those rows, and I try about 18 combinations when the truth table has five rows. I tested a few randomly chosen expressions with four and five rows. To test the masking algorithm, I tested a few expressions with six, ten and fifteen conditions, respectively.

I have tests with expressions inside **for**, **while** and **do-while** loop conditions. I've tested expressions in **if** clauses, **return** statements and assignments. To test whether the source position information is collected properly, I've tested inline functions and macros. I also have tests using member functions and member variables of C++ classes.

Before this feature is released as a part of a product, it will go through more intensive testing. Internal teams at Green Hills have started using the feature, and their feedback is being continuously incorporated to make MC/DC instrumentation more robust and bug-free. Regression tests are added whenever a new bug is discovered.

Chapter 5

Experimental Results

I discussed a few optimizations which were explored during development in sections 3.3, 3.4, 3.5.2 and 3.5.1. In the following sections, we'll explore how they impact both the space and the time overhead of MC/DC instrumentation.

5.1 Benchmark Suites

To evaluate various characteristics of my system, I used a large set of benchmark suites available at Green Hills. In the following sections, I use a small but representative subset of that to demonstrate my results and overall trends. I'll now go over each of those benchmarks and provide a brief description and some statistics. All of these suites were derived from third-party software, but had to be modified enough in order to be integrated into the Green Hills testing environment.

Some of the suites contain a single program, whereas some contain many. In the latter case, the different constituent programs generally share code, which makes it impossible to mechanically calculate aggregate statistics from combining individual test statistics.

- ***zlib*** is a compression library [9] that provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. It includes some unit tests which also serve as examples of how to use the library;

these are not meant to test the zlib library extensively. It contains approximately 6700 lines of code. When I ran this, my tools detected and instrumented 844 decisions, out of which only 242 (≈ 29%) satisfy MC/DC.

- ***stanford*** is a suite of benchmarks that are relatively short, both in program size and execution time. It contains simple, standard programs such as matrix multiplication and sorting routines. It contains approximately 600 lines of code. There were 155 boolean decisions instrumented, 135 (90%) of which satisfy MC/DC. Table 5.1 provides statistics for individual tests in this suite.

- ***specint*** is derived from benchmark suites published by the Standard Performance Evaluation Corporation (SPEC) [12]. This mostly consists of the integer benchmarks from the now obsolete CPU89 suite. Selected files from later release including CPU92 [11] and CPU95 [20] were also included. These suites were designed to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications. Among other things, *specint* includes a lisp interpreter and a PLA optimizing tool. It contains more than 10000 lines of code. This is a legacy test suite at Green Hills, and we can't run these programs anymore because of hardware compatibility issues. So I used this only for calculating space overhead by compiling the files with and without various optimizations enabled, and comparing the sizes of the resulting binaries generated.

- ***fbench*** is a floating point intensive benchmark. It contains two programs: one implementing an optical design ray tracing algorithm and one calculating the Fast Fourier Transform of a square matrix. These are self-contained programs; they can be compiled and run directly and don't need any input files or parameters. It contains around 500 lines of code. The first program has 20 boolean expressions, 12 (60%) of which satisfy MC/DC. The second program has 26 such expressions, 22 (≈ 85%) of which satisfy MC/DC. Overall, there are 45 boolean expressions here, 33 (≈ 73%) of which satisfy MC/DC.

- ***eembc2*** is a collection of the automotive, networking, office and telecom benchmarks published by the Embedded Microprocessor Benchmark Consortium (EMBC) [18]. It contains around 14600 lines of code. My program instrumented 1166 expressions in this suite, 786 ($\approx$ 70%) of which satisfy MC/DC. More detailed statistics and description are provided in the appendix.

- ***dhrystone*** is a synthetic computing benchmark intended to be representative of system programming [23]. The author collected statistics about the frequencies of different programming structures such as assignments control statements, function calls, structs, arrays, different types of loops, different operators, etc. appearing in real-life applications, and tried to recreate those statistical ratios in this benchmark. The entire benchmark suite consists of 2 .c files and a header file; the programs don't do anything meaningful and there are no tests. It contains 375 lines of code. There were only 18 expressions here, 5 ($\approx$ 28%) of which satisfy MC/DC.

- ***coremark*** is a benchmark that measures the performance of microcontrollers and CPUs used in embedded systems. Some of the algorithms implemented here include list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC (cyclic redundancy check). The programs resemble application code written by end users. Each program has one test associated with it to validate its functionality. Input files are included in the suite which are needed to test some of these programs. It contains roughly 91500 lines of code. We instrumented a total of 2326 boolean expressions, 629 ($\approx$ 27%) of which satisfy MC/DC. Table 5.1 provides statistics for individual tests in this suite.

5.2 Time Overhead

To measure the timing overhead, I used two benchmark suites; *coremark* and *stanford*. Below, we present the total number of decisions in each test of these suites and the corresponding MC/DC coverage achieved.

Test Name	Number of Decisions	Number of Decisions Satisfying MC/DC	MC/DC Percentage
coremark_pro_cjpeg2	542	127	23%
coremark_pro_core2	311	81	26%
coremark_pro_linpack2	291	53	18%
coremark_pro_loops2	385	113	29%
coremark_pro_nnet2	276	61	22%
coremark_pro_parser2	356	35	10%
coremark_pro_radix3	240	46	19%
coremark_pro_sha2	108	11	10%
coremark_pro_zip2	826	88	11%
coremark_test	117	90	77%
stanford_bubble	8	7	88%
stanford_intmm	7	7	100%
stanford_mm	7	7	100%
stanford_oscar	15	14	93%
stanford_perm	6	5	83%
stanford_puzzle	62	59	95%
stanford_queens	10	9	90%
stanford_quick	11	10	91%
stanford_towers	10	5	50%
stanford_trees	20	13	65%

Table 5.1: Coverage Information of Stanford and Coremark Test Suites

The following tables display the overhead for each test in both of those suites, and also provide aggregate statistics.

First, we consider the two optimizations on single-condition decisions. In the table below, the first column represents the overhead of an unoptimized implementation. The second column aggregates this information using geometric means for each suite. The next 3 columns describe the effect of the first optimization from 3.3, while the following 3 columns are dedicated to the impact of removing the branch condition as described in 3.4. Within each group of 3 columns, the first column shows the overhead of instrumentation, the second column shows the percentage improvement over the unoptimized version, and the third column shows the same data as the first column aggregated for the entire suite using geometric means.

Test Name	Unoptimized Instrumentation		Optimization 1			Optimization 2		
coremark_pro_cjpeg2	81.65%	75.43%	72.71%	4.92%	68.59%	34.31%	26.06%	47.58%
coremark_pro_core2	212.89%		202.92%	3.19%		151.11%	19.74%	
coremark_pro_linpack2	161.04%		139.47%	8.27%		111.80%	18.86%	
coremark_pro_loops2	21.90%		19.93%	1.61%		7.16%	12.09%	
coremark_pro_nnet2	133.06%		120.10%	5.56%		59.90%	31.39%	
coremark_pro_parser2	9.43%		7.43%	1.83%		5.91%	3.22%	
coremark_pro_radix3	14.44%		12.70%	1.53%		5.45%	7.86%	
coremark_pro_sha2	1.96%		1.73%	0.23%		1.27%	0.68%	
coremark_pro_zip2	69.51%		62.72%	4.01%		53.22%	9.61%	
coremark_test	202.53%		179.81%	7.51%		131.08%	23.62%	
stanford_bubble	147.05%	102.48%	126.65%	8.26%	92.20%	45.69%	41.03%	55.72%
stanford_intmm	189.33%		149.06%	13.92%		115.66%	25.46%	
stanford_mm	84.39%		59.84%	13.31%		49.93%	18.69%	
stanford_oscar	61.10%		56.30%	2.98%		34.29%	16.64%	
stanford_perm	10.41%		5.31%	4.63%		-21.44%	28.85%	
stanford_puzzle	236.58%		223.36%	3.93%		175.21%	18.23%	
stanford_queens	195.04%		182.45%	4.27%		183.86%	3.79%	
stanford_quick	104.49%		88.91%	7.62%		22.36%	40.17%	
stanford_towers	77.36%		73.75%	2.04%		51.32%	14.68%	
stanford_trees	37.16%		54.52%	-12.66%		16.63%	14.97%	

Table 5.2: Reduction of Time Overheads due to Single Condition Decision Optimizations

Note that the first optimization reduces overhead by roughly 10%, while the second optimization (just removing a branch instruction) causes a further 30 - 40% drop. These two optimizations bring the average time overhead down from 88% to 52%. This is a pretty significant improvement, and both of these optimizations were kept in the final version.

The next table showcases the 2 optimizations mentioned in 3.5.1 and 3.5.2. We follow the same overall syntax as in the previous table, except that the baseline to compare with is now a version of MC/DC with the previous two optimizations enabled. In each group, the first and last columns represent the absolute overhead for instrumentation, while the middle column (if present) shows the percentage improvement as compared to the baseline.

It is readily apparent that optimizing two-condition decisions is only slightly helpful; it generally has a negligible impact, but in some cases, it improves performance significantly. The overall gain is around 3%. The loop optimization is also of a similar nature, but gains from it are much more frequent as well as substantial. Taken together, these two optimizations have the potential to farther halve the time overhead

Test Name	Instrumentation with optimized DC		2-condition Optimization			Loop Optimization		
coremark_pro_cjpeg2	34.31%		36.35%	-1.52%		33.85%	0.35%	
coremark_pro_core2	151.11%		105.89%	18.01%		133.93%	6.84%	
coremark_pro_linpack2	111.80%		109.46%	1.11%		1.42%	52.12%	
coremark_pro_loops2	7.16%		7.19%	-0.03%		-3.34%	9.80%	
coremark_pro_nnet2	59.90%	47.58%	59.97%	-0.04%	42.26%	6.28%	33.53%	26.33%
coremark_pro_parser2	5.91%		4.24%	1.57%		4.63%	1.21%	
coremark_pro_radix3	5.45%		6.91%	-1.39%		-0.21%	5.37%	
coremark_pro_sha2	1.27%		1.27%	0.00%		1.27%	0.00%	
coremark_pro_zip2	53.22%		50.43%	1.82%		53.56%	-0.22%	
coremark_test	131.08%		98.38%	14.15%		95.52%	15.39%	
stanford_bubble	45.69%		45.69%	0.00%		8.09%	25.81%	
stanford_intmm	115.66%		115.66%	0.00%		0.24%	53.52%	
stanford_mm	49.93%		49.93%	0.00%		0.69%	32.85%	
stanford_oscar	34.29%		34.29%	0.00%		30.73%	2.65%	
stanford_perm	-21.44%	55.72%	-21.44%	0.00%	53.48%	3.06%	-31.19%	29.22%
stanford_puzzle	175.21%		176.48%	-0.46%		59.63%	42.00%	
stanford_queens	183.86%		145.14%	13.64%		183.72%	0.05%	
stanford_quick	22.36%		22.35%	0.01%		10.07%	10.04%	
stanford_towers	51.32%		51.22%	0.07%		52.38%	-0.70%	
stanford_trees	16.63%		16.40%	0.20%		16.26%	0.32%	

Table 5.3: Reduction of Time Overheads after Optimizing Two-condition Decisions and Loops

of MC/DC instrumentation.

5.3 Space Overhead

The space overhead was measured using a large set of benchmarks; these are used for evaluating compiler optimizations throughout the company. A total of almost 3.3 million lines of code were instrumented. I'll show the results from a small subset of these benchmarks to highlight the overall trends. The next two tables are exactly analogous to the previous two tables in 5.2, except that the numbers refer to the size of the binary generated instead of the time taken to execute it.

Observe that both of the single-condition optimizations are quite effective on all of the benchmarks; the aggregate overhead goes down from 78% to 62%. The next two optimizations are, however, almost ineffective; so they are considered mainly for improving performance and not size overhead.

Test Name	Unoptimized Instrumentation		Optimization 1			Optimization 2		
coremark	84.85%	77.89%	73.71%	6.03%	67.20%	67.94%	9.15%	61.74%
dhrystone	29.08%		24.50%	3.55%		21.91%	5.56%	
eembc2	91.22%		78.72%	6.53%		72.39%	9.85%	
fbench	49.99%		42.47%	5.01%		38.80%	7.46%	
specint	93.00%		79.04%	7.23%		72.04%	10.86%	
stanford	96.13%		82.29%	7.06%		75.32%	10.61%	
zlib	117.58%		103.20%	6.61%		95.94%	9.95%	

Table 5.4: Reduction of Size Overheads due to Single Condition Decision Optimizations

Test Name	Instrumentation with optimized DC		Two-condition Optimization			Loop Optimization		
coremark	67.94%	61.74%	65.06%	1.72%	59.29%	63.87%	2.43%	59.70%
dhrystone	21.91%		21.91%	0.00%		21.29%	0.51%	
eembc2	72.39%		68.62%	2.18%		70.53%	1.07%	
fbench	38.80%		37.05%	1.26%		37.58%	0.88%	
specint	72.04%		69.43%	1.52%		70.22%	1.06%	
stanford	75.32%		73.31%	1.14%		71.09%	2.41%	
zlib	95.94%		90.52%	2.76%		95.06%	0.45%	

Table 5.5: Reduction of Size Overheads after Optimizing Two-condition Decisions and Loops

Chapter 6

Related Work

Achieving high code coverage has always been considered important for good testing; the notion was around as early as 1963 [16]. The various metrics I described in section 1.2 are fairly standard by now [2]. Tools for computing branch and statement coverage are commonly available ([4], [1]). There has been a lot of research to optimize the instrumentation required for coverage. [22] and [6] introduce the idea of dynamic instrumentation to reduce the overhead of statement coverage. In [8], the authors combine static and dynamic instrumentation to speed up branch coverage instrumentation.

The notion of MC/DC was introduced by document DO-178B [3] in 1992, where the Federal Aviation Administration (FAA) mandated MC/DC for testing all level A (the most safety critical) software. The definition of MC/DC was later expanded to include masking MC/DC in 2001 ([7], [14]) and document DO-178C formally accepted the modified decision in 2010 [5]. The definition of MC/DC given in section 1.3 follows this masking definition. [21] gives a justification of why masking MC/DC should be accepted for certification purposes, and [13] conducts an empirical study establishing that satisfying this definition of MC/DC is not significantly harder than satisfying Condition/Decision Coverage, but it catches a wider range of bugs. However, the dependence of the MC/DC definition on source code makes it extremely sensitive to small semantically equivalent changes; [19] shows that performing MC/DC analysis on inlined vs non-inlined code leads to drastically different results.

There has been limited investigation into implementing MC/DC instrumentation. The masking algorithm described in 3.1.3 was first described in [24]. In [25], the authors used that algorithm to develop a flexible non-intrusive approach for compute metrics similar to MC/DC.

Chapter 7

Conclusion

This thesis presents the design of a system that allows programmers to visualize which parts of their code achieve modified condition/decision coverage (MC/DC) after that run their test suite. This tool is built on top of the Green Hills C/C++ compiler. I inserted static instrumentation by modifying the abstract syntax tree (AST) generated in the computer back end. I also modified some of the compiler front end code to maintain and propagate position information for each such expression. I wrote a separate utility program that interprets the extra data generated by the instrumented code and uses the associated position information to produce human readable output.

Although the main algorithms for instrumentation were already known, it was a significant engineering challenge to adapt them to an industrial compiler that has to deal with a myriad of unusual cases. Integrating my system into the Green Hills testing environment also required a fair bit of work. Finally, I had to make several benchmark-driven tweaks and optimizations to the instrumentation process in order to bring the space and time overhead down to acceptable levels. As shown in section 5.2, I was able to cut down the time overhead by almost a factor of 3.

As previously mentioned, internal teams at Green Hills Software have already started using this system. However, further testing is necessary before this system can be released to production.

Appendix A

EEMBC benchmarks

As mentioned earlier, there are 4 subcategories in our eembc benchmark suite; these roughly correspond to the AutoBench, Networking, OABench and TeleBench suites published by EMBC. These suites contain utility programs useful in specific industries, and those programs are tested extensively to ensure correctness.

In the table next page, we describe the number of boolean decisions in each individual test of these 4 suites and their MC/DC coverage statistics. Note that each row corresponds to a unique test. Some programs have only one test file associated with them whereas others have multiple tests using different datasets to test them; for example, the last 4 rows all test a Viterbi encoding program using 4 different datasets. Some of these tests require images or sound clips as input files, while others are self-contained C programs.

Test Name	Number of Decisions	Number of Decisions Satisfying MC/DC	MC/DC Percentage
eembc11_auto_a2time01	58	43	74.14%
eembc11_auto_aifftr01	55	40	72.73%
eembc11_auto_aifirf01	37	19	51.35%
eembc11_auto_aiifft01	48	33	68.75%
eembc11_auto_basefp01	37	22	59.46%
eembc11_auto_bitmnp01	185	119	64.32%
eembc11_auto_cacheb01	25	9	36.00%
eembc11_auto_canrdr01	41	23	56.10%
eembc11_auto_idctrn01	71	56	78.87%
eembc11_auto_iirflt01	79	52	65.82%
eembc11_auto_matrix01	102	70	68.63%
eembc11_auto_pntrch01	55	29	52.73%
eembc11_auto_puwmod01	92	65	70.65%
eembc11_auto_rspeed01	38	17	44.74%
eembc11_auto_tblook01	42	21	50.00%
eembc11_auto_ttsprk01	62	22	35.48%
eembc11_net_ospf_dijkstra	81	22	27.16%
eembc11_net_packet_flow_1_m	52	21	40.38%
eembc11_net_packet_flow_2_m	52	21	40.38%
eembc11_net_route_lookup	58	22	37.93%
eembc11_office_dithering	24	10	41.67%
eembc11_office_image_rotation	54	40	74.07%
eembc11_office_text_processing	70	41	58.57%
eembc11_telecom_autcor00_data_1	24	9	37.50%
eembc11_telecom_autcor00_data_2	24	9	37.50%
eembc11_telecom_autcor00_data_3	24	9	37.50%
eembc11_telecom_conven00_data_1	26	12	46.15%
eembc11_telecom_conven00_data_2	26	12	46.15%
eembc11_telecom_conven00_data_3	26	12	46.15%
eembc11_telecom_fbital00_data_1	31	14	45.16%
eembc11_telecom_fbital00_data_2	31	13	41.94%
eembc11_telecom_fbital00_data_3	31	13	41.94%
eembc11_telecom_fft00_data_1	37	13	35.14%
eembc11_telecom_fft00_data_2	37	13	35.14%
eembc11_telecom_fft00_data_3	37	13	35.14%
eembc11_telecom_viterb00_data_1	30	16	53.33%
eembc11_telecom_viterb00_data_2	30	16	53.33%
eembc11_telecom_viterb00_data_3	30	16	53.33%
eembc11_telecom_viterb00_data_4	30	14	46.67%

Table A.1: Coverage Information of EEMBC test suite

Bibliography

[1] Khalid Alemerien and Kenneth Magel. Examining the effectiveness of testing coverage tools: An empirical study. *International journal of Software Engineering and its Applications*, 8(5):139–162, 2014.

[2] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

[3] Federal Aviation Authority. Software considerations in airborne systems and equipment certification. *Document No. RTCA/DO-178B*, 1992.

[4] Ira D Baxter. Branch coverage for arbitrary languages made easy, 2002.

[5] Ben Brosgol and Cyrille Comar. Do-178c: A new standard for software safety certification. Technical report, ADA CORE TECHNOLOGIES NEW YORK NY, 2010.

[6] Kalyan-Ram Chilakamarri and Sebastian Elbaum. Reducing coverage collection overhead with disposable instrumentation. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 233–244. IEEE, 2004.

[7] John J Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical report, BOEING COMMERCIAL AIRPLANE CO SEATTLE WA, 2001.

[8] Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the sixth conference on Computer systems*, pages 199–214. ACM, 2011.

[9] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[10] SQL Developers. How sqlite is tested, 2017.

[11] Kaivalya M Dixit. New cpu benchmark suites from spec. In *COMPCON Spring 1992*, pages 305–310. IEEE, 1992.

[12] Jozo J Dujmovic and Ivo Dujmovic. Evolution and evaluation of spec benchmarks. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):2–9, 1998.

[13] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*, volume 1, pages 1B6–1. IEEE, 2000.

[14] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. A practical tutorial on modified condition/decision coverage. 2001.

[15] M Kucharski, A Trujillo, C Dunlop, and B Ahdab. Iso 26262 software compliance: Achieving functional safety in the automotive industry. Technical report, 2012.

[16] J Philip Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6:58–63, 1963.

[17] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[18] Jason A Poovey, Thomas M Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE micro*, 29(5), 2009.

[19] Ajitha Rajan, Michael W Whalen, and Mats PE Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *Proceedings of the 30th international conference on Software engineering*, pages 161–170. ACM, 2008.

[20] Jeff Reilly. A brief introduction to the spec cpu95 benchmark. *IEEE-CS TCCA Newsletter*, 1996.

[21] CAS Team et al. Rationale for accepting masking mcdc in certification projects. *Position Paper 6, Tech. Rep.*, 2001.

[22] Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient instrumentation for code coverage testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 86–96. ACM, 2002.

[23] Alan R Weiss. Dhrystone benchmark. *History, Analysis,Scores and Recommendations, White Paper, ECL/LLC*, 2002.

[24] Michael W Whalen, Mats P Heimdahl, and Ian J De Silva. Efficient test coverage measurement for mc/dc. 2013.

[25] Michael W Whalen, Suzette Person, Neha Rungta, Matt Staats, and Daniela Grijincu. A flexible and non-intrusive approach for computing complex structural coverage metrics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 506–516. IEEE Press, 2015.