

Fuzzing OpenMP Compilers

by

Raymond Chang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Raymond Chang 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

OpenMP is a widely used API for parallel programming in C/C++ and Fortran. Its flexibility and simplicity have made its usage popular in many numerical or scientific applications. The prevalence of OpenMP programs in such important areas makes its respective compiler's correctness significant. Unfortunately, OpenMP compilers are not tested as thoroughly as regular C/C++ compilers. More importantly, it is difficult to apply previous mutation-based testing techniques like EMI because of the parallelism in seed programs.

This thesis introduces new fuzz testing approaches specifically for OpenMP compilers. For existing OpenMP programs, we de-parallelize and mutate them with dead code injection and false parallelization. We also transform existing regular C programs into OpenMP programs with template-based mutations. Two test suites were used for the evaluation, the OpenMP Offloading Validation & Verification Suite (SOLLVE_VV) and programs generated from Csmith. For SOLLVE_VV and with GCC and LLVM, the proposed techniques have been shown to increase coverage by at least 4.60% and 1.81% respectively. Compared to Csmith programs, coverage is improved by at least 3.90% for GCC and 1.85% for LLVM.

Acknowledgments

I would like to thank my supervisor, Chengnian Sun for his guidance and encouragement to pursue the topics that I am passionate about. I would also like to thank my parents for their support both financially and emotionally throughout my studies. My friends have been a source of support and I would like to thank them for their wise advice. Finally, I would like to thank my girlfriend, for her patience and understanding during the duration of my studies.

Dedication

This is dedicated to my parents, who have supported me throughout my studies.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgments	iv
Dedication	v
List of Figures	viii
List of Tables	x
1 Introduction	1
2 Background	5
2.1 Fuzzing	5
2.2 OpenMP	7
2.2.1 Execution Model	8
2.2.2 Library Routines	9
2.2.3 Synchronization	10
2.2.4 Work Sharing	11
2.2.5 Single Threaded Execution	13

3	Approach	14
3.1	Preliminary Work	14
3.2	Main Approach	15
3.2.1	De-Parallelization of Seeds	16
3.2.2	Profile Generation	16
3.2.3	De-Parallelization-Based Mutation	18
3.2.4	Limitations	21
4	Evaluation	22
4.1	Experimental Setup	22
4.2	RQ1: Is Coverage Improved Over SOLLVE_VV?	22
4.3	RQ2: Is Coverage Improved Over Csmith?	24
4.4	Sample Bugs	25
5	Related Work	28
5.1	Compiler Testing	28
5.1.1	Manual Construction of Test Programs	28
5.1.2	Test Program Generation	29
5.1.3	Mutation Testing	30
5.2	Compiler Verification	31
6	Discussion	32
7	Conclusion	35
	References	36

List of Figures

1.1	OpenMP Code Examples. Figure 1.1a will result in the contents of <code>data</code> to become {1, 1, 1}. Figure 1.1b will result in the string "hello world" printed out four times on separate lines.	2
1.2	Example of an EMI Variant from Hermes.	3
1.3	Example of Guard Generation for an OpenMP Program.	4
2.1	Example of a Seed Program and an EMI Variant.	7
2.2	Format of an OpenMP Directive [20].	8
2.3	The Fork-Join Model.	8
2.4	Example of thread interleaving Figure 2.4a.	9
2.5	Example of Using a Library Routine in OpenMP.	10
2.6	Example of the <code>critical</code> Directive in OpenMP.	10
2.7	Example of the <code>barrier</code> Directive in OpenMP.	11
2.8	Example of the <code>section</code> Directives OpenMP.	12
2.9	Thread Execution of the <code>for</code> Directive.	12
2.10	Strategies to Enforce Single Threaded Execution in OpenMP.	13
3.1	Random Directive Injection.	15
3.2	An Overview of the Main Approach	16
3.3	An Example of an Environment.	17
3.4	De-Parallelization.	19
3.5	Example of Dead Code Injection.	19

3.6	Example of Template-Based Mutation.	20
3.7	False Parallelization.	21
4.1	Reduced Test Case Causing LLVM to Crash	25
4.2	Reduced Test Case with a Reduce Directive	26
4.3	Reduced Test Case that Causes OOM Error	26
4.4	Reduced Test Case that Causes LLVM's C++ compiler to Hang	27
6.1	Example of <code>std::async</code> and Possible Outputs.	33
6.2	Example of Guard Generation for an Asynchronous Program.	34

List of Tables

4.1	Coverage Improvements for SOLLVE_VV.	23
4.2	OpenMP Specific Coverage Improvements for SOLLVE_VV.	23
4.3	Coverage Improvements for Csmith.	24
4.4	OpenMP Specific Coverage Improvements for Csmith.	24

Chapter 1

Introduction

OpenMP [30] is a popular API for parallel programming. In C and C++, it adds compiler directives that simplify the process of writing parallel programs. For example, in Figure 1.1a, the `for` loop is parallelized by prepending a compiler directive. Each iteration of the loop will now be executed in parallel, allowing for a shorter runtime. OpenMP provides other compiler directives such as the one illustrated in Figure 1.1b. In this example, the compound statement will be executed in parallel by four threads. After completing the instructions within, the threads join the main thread and execution continues with a single thread. Several library routines and environment variables are also included that provide even more control. OpenMP's additions and semantics make it a flexible and simple method for writing parallel programs.

This has led to OpenMP's adoption in numerical or scientific applications where leveraging the parallel capabilities of modern hardware is crucial [17][33]. OpenMP also continues to evolve with time, with recent standards and updates to its implementations being produced. This is fuelled in part by national interests in high performance computing. For example, the *SOLLVE* project in the *Exascale Computing Project* [15] focuses primarily on improving OpenMP and its capabilities [16]. This recent interest and its use in scientific applications shows how important it is for its compilers to be correct.

```

1     int data[] = {0, 0, 0};
2     #pragma omp parallel for
3     for (int i = 0; i < 3; i++) {
4         data[i] = 1;
5     }

```

(a) Example of the `parallel for` Directive.

```

1     #pragma omp parallel num_threads(4)
2     {
3         std::cout << "hello world\n";
4     }

```

(b) Example of the `parallel` Directive.

Figure 1.1: OpenMP Code Examples. Figure 1.1a will result in the contents of `data` to become `{1, 1, 1}`. Figure 1.1b will result in the string "hello world" printed out four times on separate lines.

A common technique to ensure the quality of software such as compilers is fuzz testing. This technique relies on the production of random or semi-random test cases that are fed into the software. If a produced test case causes the software to exhibit unexpected behavior, then a bug has been detected. Production of test cases may be from scratch (generative), or rely on a corpus of existing test cases to mutate. For compiler fuzz testing, the production of test cases is an exceptionally difficult task. Test cases must be both well-defined¹ programs but still be complex enough to trigger large portions of a compiler's code base. There must also be an oracle that can determine whether a test case triggers a bug or not. Random program generators such as Csmith [44] solve these constraints by generating well-defined programs from scratch. These generators employ differential testing as the oracle. Specifically, they feed the generated programs into two separate compilers and compare their results. If the results are different or one of the compilers crashed, then a bug occurs.

Equivalence Modulo Inputs (EMI) [22] provides an alternative, mutation-based approach. EMI is a general approach for generating semantically equivalent and well-defined variants from existing test programs. An EMI variant is expected to have the same behavior as its original program for some set of inputs (not for all inputs). Any other inputs could result in behavior differing from the original program. EMI variants are ideal test

¹If a program is not well-defined, then the behavior of the program depends on the compiler not the program's semantics.

cases as their output is known (identical to original program), and they can be generated from real world programs. The original EMI-based tool, Orion [22] performed mutation only on unexecuted regions of code. Hermes [35], another EMI-based tool, builds upon Orion by allowing for mutation in executed regions of code as well. Its approach involves two stages, a profiling and mutation stage. In the profiling stage, the values of all variables at all points in a program are recorded and stored in a profile. In the mutation stage, the profiles are used to inject snippets into the program such that the behavior of the program shall be unchanged.

Figure 1.2 shows an example EMI variant from Hermes. In this example, the profiling stage would record the value of `a` as zero at all points in the program. This allows the mutation stage to generate a predicate that is always false. Any future execution with the same input would still result in the same output from the program.

```

1   int main() {
2       int a = 0;
3       for (int i = 0; i < 10; i++) {
4           // Begin inserted code
5           if (a < 0) {
6               // dead code
7           }
8           // End of inserted code
9       }
10      return 0;
11  }
```

Figure 1.2: Example of an EMI Variant from Hermes.

Challenges of Applying EMI to Test OpenMP Compilers. When attempting to apply EMI to OpenMP programs, a major issue arises. OpenMP programs are inherently parallel, and variables may be accessed by multiple threads. The values of these shared variables will differ across individual executions which excludes them from use for guard generation. This reduction in variables limits the diversity of EMI variants as mutations often require the usage of variables as in Figure 1.2. For example, in Figure 1.3 it is not as easy to generate dead snippets within the parallel region. In this example, the variable `data` is shared by two threads, which each increment the variable by 1. The value of `data` on Line 6 is non-deterministic at this point as it depends on the execution order of the threads.

```

1     int main() {
2         std::atomic<int> data = 0;
3         #pragma omp parallel for
4         for (int i = 0; i < 3; i++) {
5             // Begin inserted code
6             if (/* guard */) {
7                 ... // dead code
8             }
9             // End of inserted code
10            #pragma omp critical
11            data++;
12        }
13        return 0;
14    }

```

Figure 1.3: Example of Guard Generation for an OpenMP Program.

Our Approach. Our approach extends EMI to OpenMP programs by de-parallelizing them back into single threaded programs. With this transformation, we can ensure that the values of variables are consistent across executions. This allows us to generate EMI variants of OpenMP programs with Dead Code Injection and test compilers with them. It also allows for another technique, False Parallelization, where we surround existing statements with parallel blocks. As the blocks are not executed in parallel, the behavior of the program remains unchanged. We also introduce a third technique, Template-Based Mutation, where we transform existing C programs into OpenMP programs by inserting OpenMP templates. Conceptually, de-parallelization itself is an EMI technique, if we view the number of threads as another input of OpenMP programs. Using a single thread to run, profile, and mutate an OpenMP program significantly simplifies the task of generating equivalent program variants for testing OpenMP compilers, but to the OpenMP compilers they still need to make sure the compilation is correct under the assumption that the variants can run in multiple threads.

Our evaluation used The OpenMP Offloading Validation & Verification Suite (SOLVE_VV) [13][20] and Csmith as baselines. It found that for GCC and LLVM our techniques were able to improve coverage by at least 4.60% and 1.81% respectively. We were able to improve coverage by at least 3.90% for GCC and 1.85% for LLVM when compared against Csmith generated programs. Four bugs were also found and reported in LLVM during our preliminary attempts to develop an approach.

Chapter 2

Background

2.1 Fuzzing

Fuzzing is a software testing technique that involves providing random or semi-random input to a program and observing its behavior. It has proven to be highly effective at finding bugs and vulnerabilities in software. This is particularly important for software that is security critical. It's importance has been recognized in the software industry and companies such as Google have developed large scale fuzzing tools [3]. It has also seen interest in academia with significant amounts of research being done in the area [46].

For fuzzers, production of the input is typically guided by a set of rules or constraints. Fuzzing strategies can be divided into two categories, mutation-based fuzzing and generation-based fuzzing. In mutation-based fuzzing, the input is generated by modifying an existing input. In contrast, generation-based fuzzing generates the new input without any existing input. The existing input used in mutation-based fuzzing is called the seed.

After execution of an input, fuzzing strategies often record information about how the program behaves. This information can then be used to guide the generation of new inputs. Fuzzers that do not use this information are called black-box fuzzers. Whitebox fuzzers have complete observability of the program and its execution. Greybox fuzzers have less information but may use information such as code coverage to guide the fuzzing process [46].

For particularly important software, fuzzing strategies have been devised specifically for them. For example, compilers are a category of software where their correctness is

crucial. Bugs in compilers can lead to incorrect behavior in the software they compile. Unfortunately, compiler bugs can be difficult to identify as they may only become apparent when the compiled software is run. This has caused compiler fuzzing to become a major area of research. In mutation-based compiler fuzzing, the seed is an existing program’s source code. Mutated versions of the seed are called variants and are used by the fuzzer to test the compiler.

Bugs found upon testing a compiler include the compiler crashing, the compiler hanging, or a miscompilation. Compiler crashes or hangs are easy to determine but miscompilations are difficult to determine. Identifying miscompilation bugs requires the fuzzer to know what the expected behavior of both the seed and variant to be. There are several methods to know what the expected behavior or semantics of the seed and variant. One approach is to ensure that both the seed and its variants are semantically equivalent. However, preserving the semantics when mutating a program is difficult. Le *et al.*’s work [22] addresses this issue by introducing Equivalence Modulo Inputs (EMI).

Equivalence Modulo Inputs. Two programs that are semantically equivalent will have the same output for all inputs. EMI relaxes this definition such that they are only semantically equivalent for a set of inputs. More formally, let L denote a programming language with deterministic semantics. Given a program P in L , an execution of P with an input i can be denoted as $\llbracket P \rrbracket(i)$. As L has deterministic semantics, all executions of P for the same input i will have the same result. Let I represent a set of common inputs for two programs $P, Q \in L$. P and Q are equivalent modulo inputs w.r.t I iff $\forall i \in I, \llbracket P \rrbracket(i) = \llbracket Q \rrbracket(i)$ [35].

Figure 2.1 illustrates an example of both a seed program and an EMI variant of it. In the seed program: Figure 2.1a, the output of the program shall be five for an input of two. Executing its corresponding EMI variant: Figure 2.1b with an input of two shall still result in an output of five despite the inserted code. Note that EMI only guarantees that Figure 2.1a and Figure 2.1b shall have the same output for an input of two but not for any other input.

```

1     int main(int argc, char *argv[]) {
2         int a = atoi(argv[1]);
3         for (int i = 0; i < 3; i++)
4             {
5                 a = a + 1;
6             }
7         return a;
8     }

```

(a) A Seed Program.

```

1     int main(int argc, char *argv[]) {
2         int a = atoi(argv[1]);
3         for (int i = 0; i < 3; i++)
4             {
5                 a = a + 1;
6                 // Begin inserted code
7                 if (a == 0) {
8                     // dead code
9                 }
10                // End inserted code
11            }
12        return a;
13    }

```

(b) An EMI Variant of Figure 2.1a

Figure 2.1: Example of a Seed Program and an EMI Variant.

2.2 OpenMP

OpenMP is an API for parallel programming that is designed to be both easy to use and portable across different platforms. It adds to the C and C++ programming languages compiler directives, library routines and environment variables. Figure 2.2 illustrates the format of an OpenMP compiler directive. `directive-name` is the name of the directive and additional control can be specified using a `clause` and arguments. In Figure 1.1b, the `directive-name` is `parallel` and the `clause` is `num_threads()` with an argument of three. Compiler directives provide a straightforward way to write parallel programs. In OpenMP, parallelism means there will be multiple threads of execution. These threads are executed using the Fork-Join execution model [21].

```
#pragma omp directive-name [[,] clause[ [,] clause] ... ] new-line
```

Figure 2.2: Format of an OpenMP Directive [20].

2.2.1 Execution Model

In the Fork-Join model, the program begins with a single thread: the master thread which executes until a parallel region construct is encountered. Upon reaching the construct, the master thread is forked into multiple threads. This group of threads are known as a team. Statements within an associated parallel region are then executed in parallel by the threads. Once all forked threads have completed the statements, they terminate and the master thread resumes execution [21]. Figure 2.3 illustrates an example of the fork and join process. The statements enclosed within the curly braces starting on line 4 of Figure 2.4a are known as a structured block. In Figure 2.4a, the structured block will be marked as part of a parallel region `[]`.

Note that the order in which the statements are executed across all threads is non-deterministic. Figure 2.4 illustrates two possible outputs of the program. The first Figure 2.4b is the output if each thread finished execution of body before another. The second, Figure 2.4c is if the threads interleaved execution of the bodies.

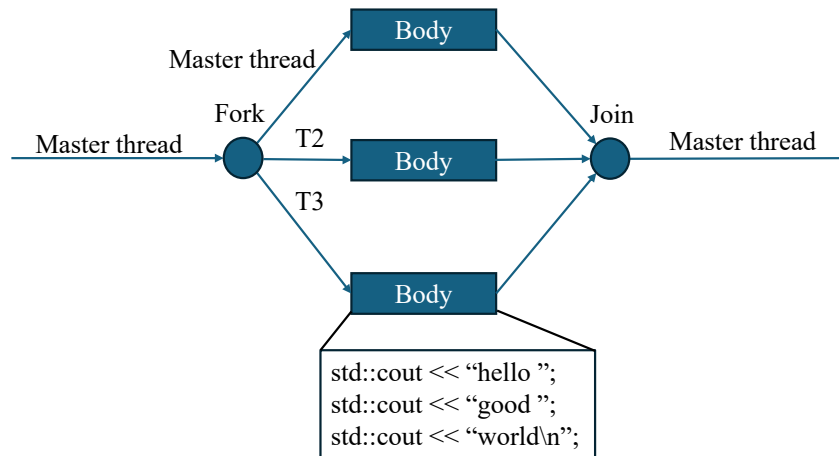


Figure 2.3: The Fork-Join Model.

```

1      int main() {
2          std::cout << "begin\n";
3          #pragma omp parallel num_threads(3)
4          {
5              std::cout << "hello";
6              std::cout << "good ";
7              std::cout << " world\n";
8          }
9          std::cout << "end\n";
10     }

```

(a) Example of the `parallel` Directive.

```

1 begin
2 hello good world
3 hello good world
4 hello good world
5 end

```

(b) Possible Output of Figure 2.4a.

```

1 begin
2 hello hello good world
3 good world
4 hello good world
5 end

```

(c) Another Possible Output of Figure 2.4a.

Figure 2.4: Example of thread interleaving Figure 2.4a.

2.2.2 Library Routines

In some cases, the added compiler directives do not provide enough control needed to write the desired parallel programs. The included library routines can be used to bridge this gap. These routines allow developers to set important parameters or query information that is useful when writing parallel programs. For example, a user can set the number of threads in a team or query how many threads are in a team with `omp_set_num_threads()` and `omp_get_thread_num()` respectively. The library routines also provide features common to parallel programming paradigms such as obtaining an identifier for a thread. Figure 2.5 illustrates an example of using the `omp_get_thread_num()` routine to set the values of an array in parallel [21].

```

1   int main() {
2       int data[] = {0, 0, 0};
3       #pragma omp parallel num_threads(3)
4       {
5           int tid = omp_get_thread_num();
6           data[tid] = 1;
7       }
8   }

```

Figure 2.5: Example of Using a Library Routine in OpenMP.

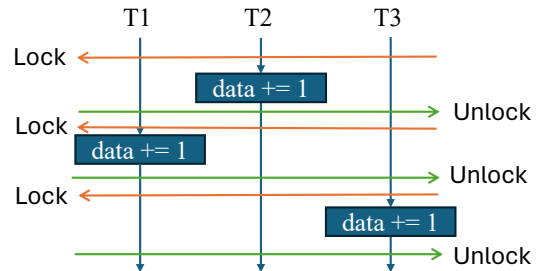
2.2.3 Synchronization

In many cases, parallel programs require synchronization between threads. One situation where this is important is when a variable is shared among threads. Access to the variable must be controlled otherwise a data race may occur. The `critical` directive can be used to ensure that only one thread accesses the variable at a time. Figure 2.6a illustrates an example of using the `critical` directive and Figure 2.6b illustrates the execution of the threads [21].

```

1   int main() {
2       int data = 0;
3       #pragma omp parallel num_threads(3)
4       {
5           #pragma omp critical
6           data += 1;
7       }
8   }

```



(a) `critical` Directive Example.

(b) `critical` Directive Thread Execution.

Figure 2.6: Example of the `critical` Directive in OpenMP.

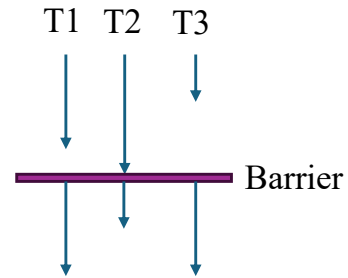
Some parallel algorithms require the programmer to ensure that all threads have reached one point of a program before continuing. The `barrier` directive can be used to achieve this. When a thread reaches the `barrier`, it will wait until all other threads have reached the `barrier` before continuing. Figure 2.7a demonstrates an example of using the `barrier` directive and Figure 2.7b illustrates the execution of the threads [21].

In Figure 2.7a, the threads will execute all statements prior to the `barrier` in parallel.

However, any thread that reaches the `barrier` will wait until all other threads have reached the `barrier`. Once all threads have reached the `barrier`, they will all resume execution.

```
1 int main() {  
2   #pragma omp parallel num_threads(3)  
3   {  
4     // statements before barrier  
5     #pragma omp barrier  
6     // statements after barrier  
7   }  
8 }
```

(a) `barrier` Directive Example.



(b) `barrier` Directive Thread Execution.

Figure 2.7: Example of the `barrier` Directive in OpenMP.

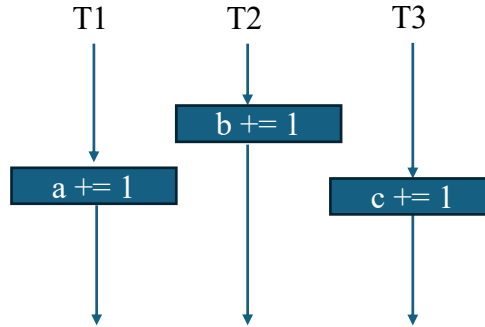
2.2.4 Work Sharing

OpenMP provides sophisticated methods to divide work among threads. For example, the `for` directive can be used to quickly parallelize a `for` loop. This directive will divide the iterations of the loop among multiple threads. An example of using the `for` directive is shown in Figure 1.1a and §2.2.4 shows the threads executing. The `sections` directive divides up the work in a parallel region among threads such that only one thread will execute a section. Figure 2.8a illustrates an example of using the `sections` directive and Figure 2.8b shows the threads executing. At the end of execution the values of `a`, `b` and `c` will be one, two and three respectively [21].

```

1 int main() {
2     int a = 0;
3     int b = 0;
4     int c = 0;
5     #pragma omp parallel sections
6     {
7         #pragma omp section
8         {
9             a += 1;
10        }
11        #pragma omp section
12        {
13            b += 2;
14        }
15        #pragma omp section
16        {
17            c += 3;
18        }
19    }
20 }

```



(a) section Directive Example.

(b) section Thread Execution.

Figure 2.8: Example of the section Directives OpenMP.

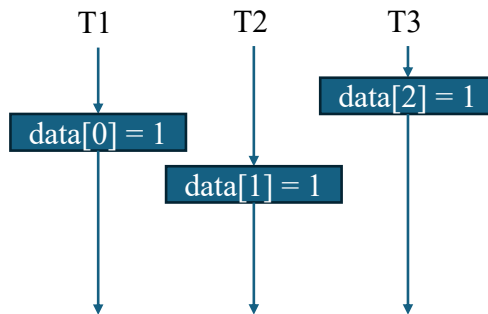


Figure 2.9: Thread Execution of the for Directive.

2.2.5 Single Threaded Execution

In OpenMP, there are several ways to enforce single threaded execution. The first is to use the `num_threads()` clause with an argument of one in the parallel directive. The second is to set the environment variable `OMP_NUM_THREADS` to one prior to execution of an OpenMP program. The third option is to use the `omp_set_num_threads()` routine with an argument of one. Both the second and third methods will set an internal variable that controls the number of threads to one. However, using a `num_threads()` clause will override the other two methods.

Figure 2.10 illustrates all three of the methods. In Figure 2.10a, the `num_threads()` clause will ensure that all statements within the structured block will execute with a single thread. For Figure 2.10b, the environment variable `OMP_NUM_THREADS` is set to one before execution. This ensures that all parallel regions will execute with a single thread. The library routine `omp_set_num_threads()` is used in Figure 2.10c to set the number of threads to one.

```
1 int main() {
2     #pragma omp parallel num_threads(1)
3     {
4         // single threaded execution
5     }
6 }
```

```
1 //Set OMP_NUM_THREADS=1
2 int main() {
3     #pragma omp parallel
4     {
5         // single threaded
6         // execution
7     }
8 }
```

(a) Using a Compiler Directive Clause

(b) Using an Environment Variable.

```
1 int main() {
2     omp_set_num_threads(1);
3     #pragma omp parallel
4     {
5         // single threaded execution
6     }
7 }
```

(c) Using a Library Routine.

Figure 2.10: Strategies to Enforce Single Threaded Execution in OpenMP.

Chapter 3

Approach

In this chapter we present both our preliminary work as well as our main approach. We first discuss our preliminary work which allowed us to better understand the challenges of fuzzing OpenMP compilers. Next we discuss our main approach which leverages our understanding of the problem to develop a novel approach for testing.

3.1 Preliminary Work

Prior to developing our existing approach, we experimented with a simpler approach so we could increase our intuition of the problem. This approach was called Randomized Directive Injection.

Randomized Directive Injection involves randomly inserting OpenMP directives into existing OpenMP programs. Note that this approach does not preserve the semantics of the program nor guarantee that the variant will be syntactically correct. Figure 3.1 shows an example of this approach. In this example we can see that the directive `#pragma omp sections` has been inserted into the source code on line 1. Note that the variant is syntactically incorrect because the inserted directive requires a structured block to be below it. Figure 2.8a shows an example of correct usage of the `#pragma omp sections` directive.

By not preserving the semantics of the program, the correct output of the variant is not easily determined. However, knowledge of the correct output is needed to determine whether a miscompilation has occurred. Another issue is that syntactically incorrect variants are unlikely to trigger deeper parts of the compiler. It is likely only the lexing and

parsing stages of the compiler would be executed. Development of this approach led to four bugs in LLVM being found and reported [7][9] [8][10]. However, its limitations led us to develop our current approach.

```
1     int data[] = {0, 0, 0};
2     #pragma omp parallel for
3     for (int i = 0; i < 3; i++) {
4         data[i] = 1;
5     }
```

(a) Source Code Before Random Directive Injection.

```
1     #pragma omp sections // Randomly inserted directive
2     int data[] = {0, 0, 0};
3     #pragma omp parallel for
4     for (int i = 0; i < 3; i++) {
5         data[i] = 1;
6     }
```

(b) Source Code After Random Directive Injection.

Figure 3.1: Random Directive Injection.

3.2 Main Approach

In this section, we present our main approach for testing OpenMP compilers. At a high level, the approach consists of four main steps and is illustrated by Figure 3.2. The first is to choose an existing program to use as a seed. The second is to profile the seed to determine which variables are deterministic and obtain their reference output. The third is to generate EMI variants of the seed. The fourth is to test the compiler with the variants and compare their output against the reference output.

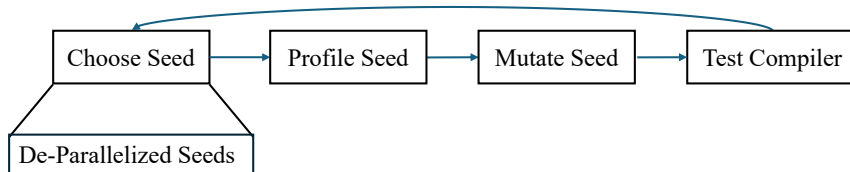


Figure 3.2: An Overview of the Main Approach

3.2.1 De-Parallelization of Seeds

The mutation stage of our approach requires that the number of threads in a parallel region is one. In some programs, the number of threads is specified with the `num_threads` clause. This occurs among programs from SOLLVE_VV but does not for Csmith generated programs as they do not have OpenMP code. We solve this by altering the value provided in the clause to one, by doing a regular expression search and replace across the SOLLVE_VV test suite. SOLLVE_VV also has some shared C header files where the number of threads is defined as a macro. We modify this value as well. These two actions combined should ensure that the number of threads is always one.

3.2.2 Profile Generation

A key requirement of our approach is to know all values of all variables at all points in the program for some set of input. This is performed by instrumenting the program with code that logs the values of variables at each point in the program. The program is then executed with the input to generate an execution profile. Figure 3.3b shows an example of a profile generated from the code on the left. The structure of a profile is an array of points such that each point is a mapping of variables to their values.

Sun *et al.* [35] formally defined an execution profile as follows: Given a program P and an input I , an execution profile $Prof$ captures all values observed at runtime for each point in the program while running P with input I . Let Var denote all in-scope variables at some program point and Loc denote the program points in P . Let Env represent a mapping from some variable $v \in Var$ to a set of its observed values. Let $Vals$ denote a variables observed values. Let a profile $Profile$ be defined as a mapping from some $l \in Loc$ to an environment Env . Figure 3.3b shows an example of a generated profile.

$$\begin{array}{ll}
 Profile = Loc \rightarrow Env & \text{(Profile)} \\
 Env = Var \rightarrow Vals & \text{(Environment)}
 \end{array}$$

In some programs from SOLLVE_VV, the initial values of variables may not be deterministic. For example, their value may be a random integer determined at runtime. We resolve this by generating multiple sets of profiles for the same program. We then compare the profiles and remove any variables that have conflicting values across profiles. Algorithm 1 illustrates the process for filtering profiles.

```

1 int main() {
2     int a = 0;
3     #pragma omp parallel
4     for (int i = 0; i < 3; i++)
5     {
6         int b = 1;
7         // do some work
8     }
9     int d = 4;
10    return 0;
11 }

```

$$Profile(7) = \left\{ \begin{array}{l} a \rightarrow \{0\} \\ i \rightarrow \{0, 1, 2\} \\ b \rightarrow \{1\} \end{array} \right\}$$

(a) An Example OpenMP Program.

(b) The Environment for Line 7.

Figure 3.3: An Example of an Environment.

Algorithm 1: Process for Filtering Profiles.

```
1 function FilterAllProfiles(Profile[] Profiles)
2   P1 := Profiles[0]
3   for P2 ∈ Profiles[1:Profiles.length] do
4     | P1 = Filter(P1, P2)
5   return P1
6 function Filter(Profile P1, Profile P2):
7   for Point ∈ P1.Points() do
8     | for Var ∈ Point.Variables() do
9       | Val = P1[Point][Var] // Get value of variable at point
10      | if P2[Point][Var] != Val then
11        | | Remove all records with Var from P1
12      | return P1
```

3.2.3 De-Parallelization-Based Mutation

Although we de-parallelized the programs with hard coded thread values, we still need to handle the case where the number of threads is not specified. We handle this by setting the `OMP_NUM_THREADS` environment variable to one. This ensures that the number of threads will be one. We then select a random subset of statements in the program to mutate. For `SOLLVE_VV` test cases, there are two types of mutations that we can apply to the statement, Dead Code Injection and False Parallelization. For Csmith generated programs, we use our Template-Based Mutation approach. Once the mutation has been applied, we can test the compiler with the variant. We then compare the output of the variant against the reference output. If the output is different, we have found a bug in the compiler. Figure 3.4 shows an example of de-parallelization. In this example only one thread is allowed to execute the `for` loop. This results in no change in the semantics of the program.

```

1      // Set the environment variable OMP_NUM_THREADS to 1
2      int[] data = {0, 0, 0};
3      #pragma omp parallel for
4      for (int i = 0; i < 3; i++)
5      {
6          data[i] = 1;
7      }

```

Figure 3.4: De-Parallelization.

Dead Code Injection. With the profiles generated, the values of variables are now known at runtime for a single input. We can leverage this information to construct random guard conditions that are always false. Figure 3.5 shows an example of this approach. New code has been inserted into the source code, but it will never be executed as the guard statement: `if (a == 1)` will always be false.

```

1      int data[] = {0, 0, 0};
2      #pragma omp parallel for
3      for (int i = 0; i < 3; i++)
4      {
5          data[i] = 1;
6      }

```

(a) Source Code Before Dead Code Injection

```

1      // Set the environment variable OMP_NUM_THREADS to 1
2      int data[] = {0, 0, 0};
3      #pragma omp parallel for
4      for (int i = 0; i < 3; i++)
5      {
6          data[i] = 1;
7          if (a == 1) {
8              data[i] = 2;
9          }
10     }

```

(b) Source Code After Dead Code Injection

Figure 3.5: Example of Dead Code Injection.

Template-Based Mutation. SOLLVE_VV only has a small number of programs we can use as seeds. In contrast, Csmith provides an unlimited number of valid programs to mutate. Unfortunately, Csmith does not generate OpenMP programs. We resolve this by injecting templates that contain OpenMP code into the programs. These templates have holes that are filled with random C code. Figure 3.6 illustrates an example of this approach. In Figure 3.6b dead code has been inserted into the source code. Within the dead code’s body, is a template that is filled in with more C code.

```

1      int data[] = {0, 0, 0};
2      #pragma omp parallel for
3      for (int i = 0; i < 3; i++)
4      {
5          data[i] = 1;
6      }

```

(a) Source Code Before Template-Based Mutation.

```

1      // Set the environment variable OMP_NUM_THREADS to 1
2      int data[] = {0, 0, 0};
3      #pragma omp parallel for
4      for (int i = 0; i < 3; i++)
5      {
6          data[i] = 1;
7          if (a == 1) { // Predicate is randomly generated
8              #pragma omp parallel
9              {
10                 // Randomly generated C code
11                 #pragma omp barrier
12                 {
13                     // Randomly generated C code
14                 }
15                 // Randomly generated C code
16             }
17         }
18     }

```

(b) Source Code After Template-Based Mutation.

Figure 3.6: Example of Template-Based Mutation.

False Parallelization. In this strategy, existing statements are wrapped in OpenMP parallel regions. As the program is single threaded, the parallel regions are always executed sequentially thus they are semantically equivalent. Figure 3.7 shows an example of this approach. In Figure 3.6b the original statement has been wrapped in a parallel region. As the program only has one thread, the parallel region is always executed sequentially.

```
1      int data[] = {0, 0, 0};
2      #pragma omp parallel for
3      for (int i = 0; i < 3; i++)
4      {
5          data[i] = 1;
6      }
```

(a) Source Code Before False Parallelization.

```
1      // Set the environment variable OMP_NUM_THREADS to 1
2      int data[] = {0, 0, 0};
3      #pragma omp parallel for
4      for (int i = 0; i < 3; i++)
5      {
6          #pragma omp parallel
7          {
8              data[i] = 1;
9          }
10     }
```

(b) Source Code After False Parallelization.

Figure 3.7: False Parallelization.

3.2.4 Limitations

There are several limitations with our approach. The first is that if an OpenMP program was designed to be run with a specific number of threads, then it is impossible to de-parallelize the program. De-parallelization would result in a program that is semantically different from the original or exhibit undefined behavior. This is especially apparent with OpenMP programs that offload processing to heterogeneous devices such as a GPU. The number of threads cannot be controlled in that situation rendering our approach unusable.

Chapter 4

Evaluation

This chapter presents the evaluation of our approach. We used the two main open-source OpenMP compilers, GCC and LLVM as test subjects.

4.1 Experimental Setup

For our evaluation we used an Intel-based machine with 20 cores and 32 GiB of ram running Ubuntu 22.04 (x86_64). GCC and LLVM were built from trunk with commit hash ids 556e7772 and 30410018 respectively. When executing the compilers, warning flags and the “-O3” optimization flag was used. Both LLVM and GCC were instrumented with their respective coverage tools. This ensures that when LLVM or GCC is executed, a coverage report will be created.

We used two sources of seed programs for the evaluation. The first source of seeds was SOLLVE_VV which had its seeds de-parallelized as described in §3.2.1. However, not all seeds could be de-parallelized as discussed in §3.2.4. The total number of seeds used for mutation was thirty. The second source was Csmith because it could provide a larger number of seed programs for evaluation. One hundred generated Csmith programs were used as seeds for mutation.

4.2 RQ1: Is Coverage Improved Over SOLLVE_VV?

In this experiment, we evaluated whether Dead Code Injection and False Parallelization could improve coverage over seeds in the SOLLVE_VV test suite. The first step was testing

the de-parallelized seeds with GCC and LLVM and collecting the coverage report. This coverage report was used as a baseline. Next, for each seed, a set number of variants were generated and used to test the compilers. Both coverage reports were then compared against each other to determine the increase in coverage. Table 4.1 shows the results for LLVM and GCC.

We also measured the increase in coverage for files that are related to OpenMP. This was done by filtering the coverage reports such that only files that contain the string “OpenMP” or “openmp” were considered. Table 4.2 shows the results.

Table 4.1: Coverage Improvements for SOLLVE_VV.

Compiler	Variants	Increase In Line Coverage	Added Lines of Coverage
LLVM	5	1.21%	27,634
LLVM	10	1.61%	36,770
LLVM	15	1.81%	41,337
GCC	5	3.20%	32,063
GCC	10	4.40%	44,087
GCC	15	4.60%	46,090

Table 4.2: OpenMP Specific Coverage Improvements for SOLLVE_VV.

Compiler	Variants	Increase In Line Coverage	Added Lines of Coverage
LLVM	5	0.15%	568
LLVM	10	0.19%	719
LLVM	15	0.58%	2,196
GCC	5	1.60%	1,885
GCC	10	2.20%	2,591
GCC	15	2.31%	2,721

4.3 RQ2: Is Coverage Improved Over Csmith?

This experiment measures whether Template-Based Mutation can improve coverage over Csmith generated seeds. Coverage reports were generated in the same way as in §4.2 except in this case; the baseline was one hundred seed programs generated by Csmith. Table 4.3 shows the coverage improvements for both GCC and LLVM. We also measured the increase in coverage for files that are related to OpenMP. We used the same method as in §4.2 and the results can be seen in Table 4.4.

Table 4.3: Coverage Improvements for Csmith.

Compiler	Variants	Increase In Line Coverage	Added Lines of Coverage
LLVM	5	1.66%	37,912
LLVM	10	1.74%	39,739
LLVM	15	1.85%	42,251
GCC	5	2.80%	28,055
GCC	10	3.90%	39,077
GCC	15	3.90%	39,077

Table 4.4: OpenMP Specific Coverage Improvements for Csmith.

Compiler	Variants	Increase In Line Coverage	Added Lines of Coverage
LLVM	5	1.65%	6,247
LLVM	10	1.65%	6,247
LLVM	15	1.67%	6,322
GCC	5	4.81%	5,666
GCC	10	5.36%	6,314
GCC	15	5.53%	6,514

4.4 Sample Bugs

During the development of our preliminary work §3.1, we found four bugs in GCC and LLVM. Two bugs caused crashes, one caused an OOM error and the last caused a timeout. When a bug was found, we used C-Reduce [32] and Perses [38] to reduce the test cases. This section provides illustrations and details on each bug.

Figure 4.1. In this issue, our testing inserted a task directive into an existing test case. Testing LLVM’s C compiler with the reduced test case resulted in a crash. The bug was reported and confirmed by a maintainer of the OpenMP implementation in LLVM. The maintainer stated that LLVM was crashing instead of emitting a message that the `task` directive was unsupported [9].

```
1 char a;
2 void foo() {
3     int b;
4     #pragma omp task
5     (char(*)[b]) a;
6 }
```

Figure 4.1: Reduced Test Case Causing LLVM to Crash

Figure 4.2. This test case was generated by mutating a test case that contained a reduction directive. Compiling the test case with LLVM’s C compiler, resulted in a crash. Reducing the test case revealed that the crash was caused by the original test case and unrelated to the mutation. The bug was reported and confirmed by the maintainers of the OpenMP implementation in LLVM. Analysis from one of the maintainers found that LLVM was crashing instead of producing an unsupported directive message [8].

```

1 int
2 foo ()
3 {
4     int r = 0;
5     #pragma omp scope reduction(+:r)      /* { dg-error "reduction
        variable 'r' is private in outer context" } */
6     r++;
7     return r;
8 }

```

Figure 4.2: Reduced Test Case with a Reduce Directive

Figure 4.3. This test case was generated by mutating a GCC test and triggered an Out of Memory bug in LLVM’s C compiler. After reducing the test case, we discovered that the bug was unrelated to the mutation. The bug was reported and marked by a maintainer as having the same underlying issue as another bug [7].

```

1 static char * name[] = {
2     [0x80000000] = "bar"
3 };

```

Figure 4.3: Reduced Test Case that Causes OOM Error

Figure 4.4. For this test case, our mutation strategy inserted a `critical` directive into a GCC test case. When testing LLVM’s C++ compiler with the mutated test case, the compiler would hang. The bug only affects LLVM’s C++ compiler but not its C compiler [10].

```

1 typedef struct {
2     int e; int f; int g; int h; int i; int j;
3     int k; int l; int m; int n; int o; int p;
4 } Scl16;
5
6 Scl16 g1sScl16, g2sScl16, g3sScl16, g4sScl16, g5sScl16, g6sScl16,
   g7sScl16,
7 #pragma omp critical
8     g8sScl16, g9sScl16, g10sScl16, g11sScl16, g12sScl16, g13sScl16,
   g14sScl16,
9     g15sScl16, g16sScl16;
10
11 void testvaScl16();
12
13 void
14 testitScl16() {
15     testvaScl16(g10sScl16, g11sScl16, g12sScl16, g13sScl16, g14sScl16,
16     g1sScl16, g2sScl16, g3sScl16, g4sScl16, g5sScl16, g6sScl16,
17     g7sScl16, g8sScl16, g9sScl16, g10sScl16, g11sScl16, g12sScl16,
18     g13sScl16, g14sScl16, g15sScl16, g16sScl16);
19 }

```

Figure 4.4: Reduced Test Case that Causes LLVM's C++ compiler to Hang

Chapter 5

Related Work

Our approach is related to existing work on improving the correctness of compilers. In this section we present related work in the areas of compiler testing and compiler verification.

5.1 Compiler Testing

Compiler testing has been an active area of research for many years. Strategies have ranged from specialized techniques specifically for one programming language to more general approaches. Related areas such as understanding compiler bugs and evaluating the testing strategies themselves have also been explored [42] [37].

A key part in compiler testing is the generation of test programs. Chen *et al.* [11] classified program generation techniques into three categories: manual construction of test programs, test program generation and mutation testing.

5.1.1 Manual Construction of Test Programs

The simplest approach to testing a compiler is to manually write test programs. This can take the form of unit tests or regression suites that contain a collection of test programs. Existing compiler projects such as GCC and LLVM have extensive test suites and even their own testing infrastructure [18] [27]. Another source of test cases is SOLLVE_VV which was designed specifically to validate OpenMP compilers.

While manual test suites are useful for verifying the correctness of certain aspects of a compiler, they rely on a developer to write them. This is costly in terms of developer

time and is limited by the creativity of the developer. In comparison, our approach uses automated techniques that can generate large amounts of unique test cases with minimal developer effort.

5.1.2 Test Program Generation

Generating random test programs is a popular research direction in compiler testing. Program generators have been created for many programming languages ranging from popular languages such as C++ to more specialized languages such as GLSL. Approaches to generating random test programs can be categorized into three groups: Grammar-directed, Grammar-aided and Other approaches [11].

Grammar-directed Approaches. Grammar-directed approaches generate programs by using the programming language’s grammar. A common approach among them is to traverse the production rules of the grammar to generate parts of the test program [11]. Purdom’s work [31] [11] is an example of this approach and was used to test automatically generated parsers. Burgess and Saidi [6] present a method for testing optimizing Fortran compilers.

Grammar-aided Approaches. Grammar-aided approaches use both the grammar of a programming language and heuristics to guide the generation of programs. They often start with a template of a program that is filled in by leveraging the grammar [11]. Sirer *et al.* [34] leverage this approach to test Java implementations. Yang *et al.*’s Csmith was designed specifically to test C compilers. The programs generated by Csmith contain complex control flow and most features in the C language. However, the most important feature is that none of the generated programs have undefined or unspecified behavior. Variants of Csmith have also been created to test C-like languages. For example, CLSmith [26] is a tool that generates test programs for OpenCL compilers. OpenCL is a programming language for parallel programming across different types of hardware such as CPUs, GPUs or other specialized accelerators [41] [40].

Other Approaches. Some approaches do not use the grammar of a programming language at all. In Berry’s [5] [11] approach, the frequency of language feature use is leveraged to generate test programs. Randprog, [14] a predecessor to Csmith, was designed specifically for finding miscompilations involving the `volatile` keyword in C. Nagai *et al.* [29] introduces an approach to test arithmetic optimization passes in C compilers. Dfusor [43] focuses on how debug information generation affects the compiler output. Zhang *et al.* [45] propose the skeleton program enumeration problem and utilize their approach

to test C/C++ compilers.

Compared To Our Approach. While the approaches mentioned above are capable of generating test programs that target C/C++ compilers, they are not designed to generate OpenMP programs. CLSmith is the only tool mentioned above that is capable of generating parallel programs but it also cannot produce OpenMP programs.

5.1.3 Mutation Testing

Mutation testing is a technique that changes existing programs to create new ones. Chen *et al.* [11] classify these techniques into two categories: Semantics-preserving Mutations and Non-semantics-preserving Mutations.

Semantics-preserving Mutations. Equivalence Modulo Inputs (EMI) is the basis of most semantics-preserving mutation testing techniques for compilers [11]. In EMI, the definition of semantic equivalence between two programs is relaxed. Two programs are considered semantically equivalent if they produce the same output for only a given set of inputs. This is in contrast to the traditional definition of equivalence, where two programs are equivalent if they produce the same output for all possible inputs [22]. In the first EMI approach: Orion [22], mutants are created by profiling a program on one input I and modifying unexecuted (dead) regions of code. The mutant and the original are then run on I and it is expected that their outputs be identical. Athena [23], extends Orion by using a distance metric to maximize the difference between the mutant and original program.

Hermes [36], is the first EMI-based tool that can create mutants by injecting statements in live (executed) code. It extends the profiling done by Athena, to include the values of integer variables. Based on these values, it synthesizes three types of snippets and injects them into live code regions. In Lidbury *et al.*'s [26] work, they implement EMI for OpenCL compilers. They note that existing OpenCL kernels rarely have dynamically dead code and overcome this issue by injecting *dead-by-construction-code*. Sun *et al.*'s work This concept is similar to the *False Conditional Block* in [36] as the injected code is dynamically dead. It differs from Hermes' in that the guard in the dynamically dead code is not generated based on profiling information but rather a parameter passed to the OpenCL kernel.

Non-Semantics-preserving Mutations. Some approaches mutate existing programs but do not preserve the original program's semantics. Chen *et al.*'s coverage-directed approach for testing JVM implementations [12] utilizes differential testing. LangFuzz [19] is a tool that uses parts of programs that have already caused invalid behavior. It has

been used to test Mozilla Firefox’s JavaScript interpreter [19]. Sun et al’s [35] work uses differential testing to find compiler warning defects.

Compared to Our Approach. Our work builds off of Hermes and adapts EMI for OpenMP compilers. We add three main changes to Hermes’ approach. The first is that we add a filtering step to the profiling stage. The second is we introduce De-Parallelization-based mutations. The third is the two new mutations we introduce, False Parallelization and OpenMP Template-Based Mutation.

5.2 Compiler Verification

Compiler verification involves writing a formal proof of correctness for a compiler [36]. CompCert [24] [25] is the most famous verified optimizing compiler and supports the C language. Its proof of correctness was done with the Coq proof assistant [39]. Barrière *et al.*’s [4] work reuses CompCert and its proofs for a JIT compiler. Mansky and Du [28] extend CompCert and its proofs to verify C programs.

Compared to Our Approach. Compiler verification is a powerful technique for ensuring the correctness of a compiler. However it is time consuming to write a proof of correctness and requires expertise in formal methods. Our approach has neither of these requirements but cannot assure the total correctness of a compiler like compiler verification.

Chapter 6

Discussion

Asynchronous functions provide a method to run instructions concurrently. In popular programming languages, function calls are usually synchronous and block the caller until the function returns. In contrast, asynchronous functions immediately return to their caller but continue to perform their work concurrently [2]. This allows the caller to execute other instructions while the asynchronous function is running. Asynchronous functions can be particularly useful when a program needs to perform expensive I/O operations. This allows for other instructions to be executed while waiting for the I/O operations to complete.

In C++, the `std::async` function template provides a way to run a function asynchronously and returns a `std::future` object. `std::future` can be used to retrieve the return value of an asynchronous function call. Figure 6.1 shows an example of using `std::async`. In Figure 6.1a, two functions, `print_hello` and `print_world`, are executed asynchronously. Then on line 14, an exclamation mark is printed and later on, the return value of `print_hello` is printed. The possible outputs of this program are shown in Figure 6.1b. Note that the order of the output is non-deterministic because `print_hello` and `print_world` may be executed concurrently. Asynchronous programming may also be done in other languages such as Java. Java provides the `Future` interface to represent the return value of an asynchronous computation [1].

```

1 int print_hello() {
2     std::cout << "hello ";
3     return 1;
4 }
5
6 void print_world() {
7     std::cout << "world ";
8 }
9
10 int main() {
11     // Run print_hello and print_world concurrently
12     auto fut1 = std::async(print_hello);
13     auto fut2 = std::async(print_world);
14     std::cout << "! ";
15     // Print out the return value of print_hello()
16     std::cout << fut1.get();
17 }

```

(a) Example of `std::async` in C++.

```

1 // Possible Output 1:
2 hello ! world 1
3 // Possible Output 2:
4 ! hello 1world

```

(b) Possible Outputs of Figure 6.1a.

Figure 6.1: Example of `std::async` and Possible Outputs.

When attempting to apply EMI to asynchronous programs, we may encounter the same issues as with OpenMP programs. If a variable is shared between multiple asynchronous function calls their values may differ across individual executions. Similarly with OpenMP, this would exclude them from being used for guard generation. Figure 6.2 shows an example of guard generation for an asynchronous program. The variable `x` is shared between three asynchronous calls to the lambda function `foo`. The value of `x` is non-deterministic at line 4 as it depends on the execution order of the asynchronous functions. This excludes it from use for guard generation.

```

1 int main() {
2     std::atomic<int> x = 0;
3     auto foo = [&x](int y) {
4         if ( /* guard */ ) {
5             ... // dead code
6         }
7         if (y == 1) {
8             x--;
9         } else {
10            x++;
11        }
12    };
13    // Run foo asynchronously three times
14    auto fut1 = std::async(foo, 0);
15    auto fut2 = std::async(foo, 1);
16    auto fut3 = std::async(foo, 0);
17 }

```

Figure 6.2: Example of Guard Generation for an Asynchronous Program.

In theory, this issue may be solved similarly to de-parallelization by forcing all asynchronous calls to become synchronous. This would ensure that the order of execution is deterministic and prevent any parallelism or concurrency in the program. However, in practice, popular asynchronous runtime systems do not provide a simple method to force all asynchronous calls to be synchronous. There is also not usually a method to control the number of threads, preventing us from making the program non-parallel.

Chapter 7

Conclusion

This thesis presents the first approach designed specifically to test OpenMP compilers. It builds upon existing EMI-based work to generate new variants of seed programs. When mutating existing OpenMP programs, we de-parallelize OpenMP programs and use two strategies to generate variants. The first is to insert dead code snippets into the program and the second is to falsely parallelize existing code. When mutating Csmith generated programs, we convert them to OpenMP variants by inserting dead OpenMP templates.

Our evaluation has shown our approach to be effective in improving coverage over existing work. Compared to SOLLVE_VV we have shown that our approach can improve coverage of GCC and LLVM by at least 4.60% and 1.81% respectively. For Csmith generated programs, our approach has improved coverage by at least 1.85% for LLVM and 3.90% for GCC. Preliminary attempts to develop this approach also led to four bugs being found and reported in LLVM. Due to time constraints, an extensive fuzzing campaign was not run. In our future work, we plan to apply our techniques to perform long-term fuzz testing on GCC and LLVM.

References

- [1] Java Development Kit Version 17 API Specification. <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Future.html>.
- [2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.
- [3] Abhishek Arya, Oliver Chang, Jonathan Metzman, Kostya Serebryany, and Dongge Liu. OSS-Fuzz, May 2024.
- [4] Aurèle Barrière, Sandrine Blazy, and David Pichardie. Formally Verified Native Code Generation in an Effectful JIT: Turning the CompCert Backend into a Formally Verified JIT Compiler. *Proceedings of the ACM on Programming Languages*, 7(POPL):249–277, January 2023.
- [5] Daniel M. Berry. A new methodology for generating test cases for a programming language compiler. *ACM SIGPLAN Notices*, 18(2):46–56, February 1983.
- [6] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38(2):111–119, January 1996.
- [7] Raymond Chang. Clang trunk OOM on GCC test case · Issue #77690 · llvm/llvm-project. <https://github.com/llvm/llvm-project/issues/77690>.
- [8] Raymond Chang. [Clang][OpenMP] Clang crashes with OpenMP reduction pragma · Issue #77535 · llvm/llvm-project. <https://github.com/llvm/llvm-project/issues/77535>.
- [9] Raymond Chang. [Clang][OpenMP] clang trunk crashes with OpenMP task directive · Issue #74412 · llvm/llvm-project. <https://github.com/llvm/llvm-project/issues/74412>.

- [10] Raymond Chang. Incorrectly placed OpenMP pragma results in timeout in clang++ trunk · Issue #78631 · llvm/llvm-project. <https://github.com/llvm/llvm-project/issues/78631>.
- [11] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A Survey of Compiler Testing. *ACM Computing Surveys*, 53(1):1–36, January 2021.
- [12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, Santa Barbara CA USA, June 2016. ACM.
- [13] Bronis R. De Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, editors. *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings*, volume 11128 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2018.
- [14] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software*, pages 255–264, Atlanta GA USA, October 2008. ACM.
- [15] Exascale Computing Project. Home Page. <https://www.exascaleproject.org/>.
- [16] Exascale Computing Project. SOLLVE. <https://www.exascaleproject.org/research-project/sollve/>.
- [17] Yehonatan Fridman, Guy Tamir, and Gal Oren. Portability and Scalability of OpenMP Offloading on State-of-the-art Accelerators, May 2023.
- [18] GCC team. Installing GCC: Testing. <https://gcc.gnu.org/install/test.html>.
- [19] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. *Proceedings of the 21st USENIX conference on Security symposium*, August 2012.
- [20] Thomas Huber, Swaroop Pophale, Nolan Baker, Michael Carr, Nikhil Rao, Jaydon Reap, Kristina Holsapple, Joshua Hoke Davis, Tobias Burnus, Seyong Lee, David E. Bernholdt, and Sunita Chandrasekaran. ECP SOLLVE: Validation and verification testsuite status update and compiler insight for OpenMP. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 123–135, 2022.

- [21] Blaise Barney Laboratory, Lawrence Livermore National. OpenMP. <https://hpc-tutorials.llnl.gov/openmp/>.
- [22] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, Edinburgh United Kingdom, June 2014. ACM.
- [23] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 386–399, Pittsburgh PA USA, October 2015. ACM.
- [24] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, Charleston South Carolina USA, January 2006. ACM.
- [25] Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363–446, December 2009.
- [26] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, August 2015.
- [27] LLVM Project. LLVM Testing Infrastructure Guide — LLVM 19.0.0git documentation. <https://llvm.org/docs/TestingGuide.html>.
- [28] William Mansky and Ke Du. An Iris Instance for Verifying CompCert C Programs. *Proceedings of the ACM on Programming Languages*, 8(POPL):148–174, January 2024.
- [29] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random Testing of C Compilers Targeting Arithmetic Optimization. *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2012.
- [30] OpenMP Architecture Review Board, Michael Klemm, and Bronis R. de Supinski, editors. *OpenMP Application Programming Interface Specification Version 5.0*. Independent, Independently Publishing, first printing edition, 2018.
- [31] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, September 1972.

- [32] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-Case Reduction for C Compiler Bugs.
- [33] Ronald W. Shonkwiler and Lew Lefton. *An Introduction to Parallel and Vector Scientific Computation*. Cambridge University Press, 1 edition, August 2006.
- [34] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. *ACM SIGPLAN Notices*, 35(1):1–13, January 2000.
- [35] Chengnian Sun, Vu Le, and Zhendong Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering*, pages 203–213, Austin Texas, May 2016. ACM.
- [36] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 849–863, New York, NY, USA, October 2016. Association for Computing Machinery.
- [37] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 294–305, Saarbrücken Germany, July 2016. ACM.
- [38] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 361–371, Gothenburg Sweden, May 2018. ACM.
- [39] The Coq Team. Welcome! | The Coq Proof Assistant. <https://coq.inria.fr/>.
- [40] The Khronos Group Inc. The OpenCL™ Specification.
- [41] The Khronos Group Inc. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. <https://www.khronos.org//>, July 2013.
- [42] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, pages 4873–4882, Macau, SAR China, August 2023. International Joint Conferences on Artificial Intelligence Organization.

- [43] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 146–158, New York, NY, USA, January 2023. Association for Computing Machinery.
- [44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *ACM SIGPLAN Notices*, 46(6):283–294, June 2011.
- [45] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, Barcelona Spain, June 2017. ACM.
- [46] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys*, 54(11s):1–36, January 2022.