Finding Missing Automatic Vectorization Opportunity by Differential Testing
差分テストを用いた自動ベクトル化機会の自動探索

by

Kohei Asano
浅野 光平

A Master Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
on January 17, 2024
in Partial Fulfillment of the Requirements
for the Degree of Master of Information Science and Technology
in Computer Science

Thesis Supervisor: Shinya Takamaeda　高前田 伸也
Associate Professor of Computer Science

ABSTRACT

In recent years, the importance of parallel computing has increased for many applications. This paper focuses on effectively utilizing vector extension instructions, which CPU vendors actively develop. The effective use of vector extension instructions significantly contributes to accelerating computations in deep learning. For example, in the latest programming language, Mojo, effective utilization of SIMD instructions has accelerated the performance of some benchmarks by up to 30,000 times compared to Python.

Effective utilization of the current vector extension instructions requires a deep understanding of hardware and ISA. Although there are methods involving automatic vectorization by compilers, they cannot yet achieve the performance levels of manual optimization. The most significant difficulty in automatic vectorization in the compiler lies in the high management cost due to the diversity of hardware-dependent vector extension instructions.

In this paper, we propose a novel method to find a missing opportunity in automatic vectorization, based on test generation and differential testing to automatically discover bugs and automatic vectorization optimization opportunities in compiler infrastructure. For fully automated Automatic Vectorization management, we have developed a random C program generator, CSmith-based test generator for automatic vectorization suitable for vectorization, and have shown that it can efficiently insert optimization opportunities for target-independent vectorization and that the tool is scalable. We have also used it to generate tests for different ISAs. We also developed a comparison framework for comparing machine programs generated for different ISA targets using LLVM statistics. By comparing machine programs compiled from the same program for different ISA targets, we confirmed that the difference in Automatic Vectorization performance can be measured, and showed that it can be used as a component of the automatic discovery of complementary Automatic Vectorization opportunities for ISA.

論文要旨

近年, 様々なアプリケーションで並行計算の重要性が増している. 本研究では CPU のベンダーにより活発に開発されているベクトル拡張命令の有効活用に注目する. ベクトル拡張命令の活用は, 深層学習などの計算の高速化に大きく寄与し、最新のプログラミング言語 Mojo では SIMD 命令の有効活用により一部のベンチマークで Python のパフォーマンスを最大 30000 倍以上高速化した事例もある.

現在のベクトル拡張命令の有効活用にはハードウェアや ISA への深い理解が必要になり, コンパイラの自動ベクトル化による方法もあるが, 手動の高速化ほどのパフォーマンスは実現できない. 自動ベクトル化の難しさの最大の要因としては, ハードウェアに大きく依存するベクトル拡張命令の多様性による管理のコストの高さがあげられる.

本論文では, 近年注目されているテスト生成と差分テストを組み合わせたコンパイラ基盤のバグや最適化機会の自動探索をフレームワーク化し, それを用いた自動ベクトル化に特化したバグ/最適化機会を自動で発見する手法を提案する. 本研究の貢献は 1. ベクトル化の差分テストに特化したテストの生成手法の提案, 2. 異なるハードウェアターゲット向けのコンパイルの差分の設計である. 差分テストの自動化のために, 私たちはランダムな C プログラム生成ツール, CSmith に基づいた, ベクトル化に適した自動ベクトル化向けのテスト生成ツールを開発し, ターゲット非依存なベクトル化に対する最適化機会の挿入を効率的に行えることとそのツールの拡張可能性を示した. また, それを用いて異なる ISA ターゲットに対して生成された機械プログラムを LLVM の統計を利用して, 比較するための比較フレームワークを開発した. 同一のプログラムから異なる ISA ターゲット向けにコンパイルされた機械プログラムを比較することで, 自動ベクトル化のパフォーマンスの差が計測可能なことを確かめ, ISA の相補的な自動ベクトル化の機会の自動発見のコンポーネントとして利用できることを示した.

Acknowledgements

I truly appreciate all the help my supervisor Associate Professor Shinya Takamaeda and my former supervisor Professor Naoki Kobayashi gave. They gave me a lot of advice and encouragement. I would also like to thank the secretary Fumiko Yamaura for arranging the best environment for me to concentrate on my research. Thanks to the best environment they created, I could have discussed my work with my laboratory members both in Kobayashi and Takamaeda Laboratory, and I must thank them for their valuable advice and encouragement. I also appreciate my family's overall support.

Contents

Chapter 1

Introduction

In recent years, various applications such as machine learning, image processing, and speech processing have attracted much attention in various areas, such as large-scale language modeling, automated driving, and so on, where parallel processing can improve execution speed. Although various parallel processing paradigms have been investigated in software and hardware, CPU vendors have devoted many resources to improving parallelism using vector extension instruction sets. And their effective use is essential, as we see recently, Intel has announced a new vector extension instruction set AVX-10[1], and Arm has developed a variable-length vector instruction called Scalable Vector Extension (SVE)[2]. If they are used well, programs can be accelerated and small in size. For example, a recent programming language, Mojo, achieved over 30000 times speedup compared to Python[3] by effectively leveraging SIMD instructions in a way of designing an intermediate language that has high affinity with the polyhedral model for a certain kind of benchmark, like Mandelbrot set computations.

However, for other scientific or more general tasks in application development in other programming languages such as C, C++, and Rust, the development of software using vector extension instructions is as difficult as any other parallelism and inevitably creates hardware dependence in compiler implementation, which is a difficulty that requires an understanding of the hardware and results in software with poor portability. More concretely, it is necessary to write the assembly languages directly, a program using intrinsic instructions provided by the processor vendor, use the compiler's automatic vectorization, or use a third-party library that already uses them well, depending on the application. Among these, automatic vectorization of compilers should eliminate the need for other methods if sufficient performance is achieved. Still, it has not yet gained enough performance compared to other utilization methods. One of the most significant difficulties in vector extension utilization is the target dependence mentioned above.

As an example of hardware dependence, the differences in types of vector extension operations exist. Still, there are also differences in handling register size among the different ISAs. For example, Intel's x86 vector extension instruction set uses fixed-length vector registers. In contrast, Arm's SVE uses variable-length vector registers for portability, although the length varies depending on the target machine. The vector extension instruction RVV specified in RISC-V, an open-source ISA that has been actively developed in recent years, also defines registers with the size of vector registers and can handle variable-length vector registers. Although each vendor employs contributors to the open-source compiler infrastructure, they focus on developing their targets, making their implementations distributed and complicated. A look into review systems such as LLVM shows

that many patches are created for different targets, and the development cost is high.

Solutions to this diverse vectorizer management problem have been actively researched in recent years. The most representative examples are VeGen[4] and Minotaur[5]. VeGen inputs ISA manual like Intel's Software Developer Manual (SDM) and automatically generates a C++ Vectorizer program that handles ad hoc peephole optimization on the LLVM middle-end. Minotaur[5], as known as SIMD-oriented Superoptimizer, performs an enumerative automatic search and finds the profitable transformation by using LLVM Machine Code Analyzer, and verifies the validity with the LLVM IR transformation validator, alive2[6]. However, the former still isn't used for practical compiler development such as GCC or LLVM because its generated code isn't manageable for LLVM developers with the original implementation, and the latter didn't perform well enough because of the complex burden of enumeration search with formal validation.

This paper focuses on automatic vectorization opportunities and bug detection in the effective use of vector extension instructions and proposes a method that combines automatic test generation and differential testing. Test generation using CSmith[7] or other smith tools[8, 9] is a famous approach for practical development improvement research. And various Fuzzing for the compiler toolchains is proposed[10, 11, 12]. They automatically find the performance regressions, or bugs introduced by contributors' patches, and create a smart report by bisecting the generated test or commits. So vectorizer automatic fuzzer would alleviate the difficulty of automatic vectorization management issues.

To create a fully automated Vectorizer Fuzzer that increases Vectorizer performance and development efficiency, the design must be made to do the following automatically.

1. Test Generation
2. Differential Testing
3. Analysis for found alert

Test generation means producing the input program that characterizes the feature to be tested; Differential Testing is the way to find the missed opportunities/bugs/regressions, and Analysis is to reduce the test to understand the root cause found easily. These three items can apply to various kinds of differential test-based fuzzing methods for compilers. This paper proposes Test Generation and Differential Testing with particular effectiveness for target-independent LoopVectorizer.

This study's key contributions are as follows.

1. A novel method to find a missing opportunity in automatic vectorization, based on test generation and differential testing, to automatically discover bugs and automatic vectorization optimization opportunities in compiler infrastructure.
   (a) A test generation method specialized for vectorization opportunities, based on random C program generator CSmith[7] and Clang AST Transformer.
   (b) A comparison for comparing vectorizer performance with different hardware targets exploiting LLVM compiler optimization statistics.
2. An empirical analysis of the method to automatically find and analyze missing opportunities or bugs found or not found by the method.

The remainder of this paper is organized as follows. Section 2 describes the background for compiler organization, vectorizer, vector extensions, and recent compiler fuzzing with moderate related work. Section 3 describes the proposed Test Generation method, Vectorizer Mutator, and Differential Testing method Vectorizer Fuzzer. Section 4 describes the evaluation of the proposed Vectorizer Mutator and Vectorizer Fuzzer, and discussions with future work. Section 5 describes the related work. Section 6 concludes this paper with future directions.

Chapter 2

Background

In this chapter, we briefly review the design of compiler optimization with intermediate representations of the Vector extension and recent compiler fuzzing research. In particular, various methods and target applications have emerged in recent years for Fuzzing the compiler toolchain, and this chapter will attempt to explain them in terms of a common framework.

2.1 Vector Extension Instructions

This paper defines vector extensions as instructions that handle multiple data on single instructions specified on optional ISA extension specs for parallel data processing. From the perspective of parallel computing, computer architectures were classified into four major categories according to Flynn's taxonomy: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). SIMD Instructions handles arrays or so-called *Vector* e.g. 8 single precision floating points for 256-bit vector registers, or 16 4-byte integers for 512-bit vector registers. Each component of the vector register is called *Lane* for SIMD register or instructions and it's used as a basic component for Target-independent Automatic Vectorization algorithms. SISD and SIMD on Flynn's taxonomy are shown in Fig. 2.1. SIMD instructions also equal vector extensions, an optional ISA extension for parallel data processing. Vector extensions have recently been developed to focus more on domain-specific instructions. AVX512-VAES for AES computation, AVX512-GFNI for a Redundant Array of Independent Disk (RAID), and AVX512-VNNI for Neural Network calculations represented by dot products. However, some researchers have distinguished vector extensions from SIMD because new lane interchanging instructions[4], like dot products instructions used on Neural Network computations, seem to have multiple operations, not a single operation. Let us see an example of such instructions on the x86 AVX-512 and Arm SVE, focused instruction sets on this paper. SSE instruction, a vector extension for x86, has a single instruction ADDSUBPD with semantics like Figure 2.2. Still, it is called non-SIMD in [4] because two operations, addition, and subtraction, are performed for each lane interchangeably. Since the optimization for such instructions is not fully handled by the generalization of the SLPVectorizer described later, such lane interchanging cases are handled by ad-hoc peephole optimization. On Armv8 SVE, we can port ADDSUBPD with ADDHNB (Add narrow high part) and SSUBWB (Signed subtract wide).

Another example of domain-specific vector extension instruction on x64 is the VPDPWSSD 2.3, its lane size operation is not on Arm SVE, but a variant, VPDPBUSD, corresponds with sdot in Armv8 SVE. That is x86's dot product in-

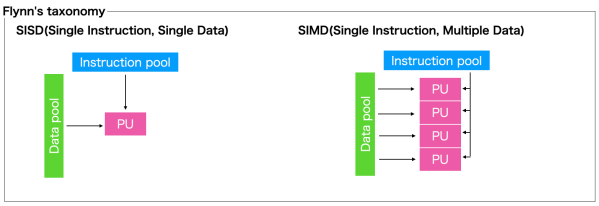

Figure 2.1: SIMD and SISD on Flynn's taxonomy

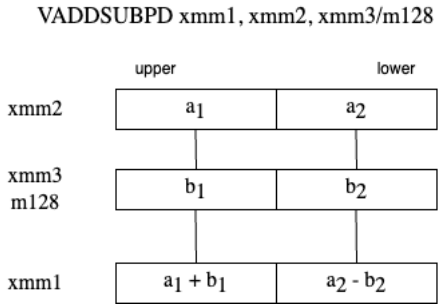

Figure 2.2: Lane Diagram for VADDSUBPD Reproduced from [13]

structions for neural network computations on AVX512-VNNI (Advanced Vector eXtension Vector Neural Network Instructions) instructions.

2.2 Difficulty on Vector Extension Instructions

As described above, instruction sets of major architectures, e.g., AArch64 Neon and SSE2, correspond on each instruction kind, but some instructions, such as PMOVMSKB on SSE, are absent on Neon. Also, for vector widths and their specifications, they are dispersed. In particular, regarding fixed or variable vectors that increase the compiler's automatic vectorization implementation's complexity, we will discuss those with fixed-length vectors and instruction sets with variable-length instructions. Variable length vector operations are proposed for programs having portability for scalability, especially in supercomputing. Fixed length vector extensions are Arm's NEON, x86's SSE, and AVX series vector extensions. The representative for variable length vector extension is the Scalable vector extension of the Arm, also used in Japanese Super Computer Fugaku, and RISC-V RVV is a variant for variable length vector extension in the point using the register to set the dynamic length for vector operations.

2.3 Compiler Organization

Modern programming language processing systems implementations are designed using intermediate representations to divide optimization and verification functions into separate modules, as LLVM[14, 15] does. This creates independence of the developer's responsibilities and avoids implementation fragmentation. However, at the same time, it makes integrated behavior complex and challenging to analyze the causes of regression/bugs. Whether the division into modules is good or not is a question of subtlety, and some people prefer to integrate them[16]. For example, Instruction Selection, the conversion from an independent intermediate representation to a target-dependent intermediate representation, is also changed from Basic Block unit implementation to Global implementation from

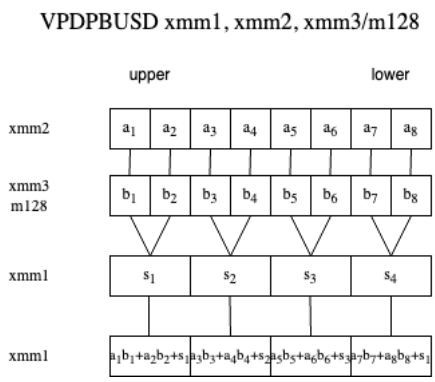

Figure 2.3: Lane Diagram for VPDPWSSD (AVX512-VNNI) Reproduced from [13]

the viewpoint of compile time. [17] Against this background, the management and performance tuning of compiler optimization is a complicated problem, and black-box analysis methods using Fuzzing[10, 11, 12, 7] and other methods are being studied, as described later. The compiler optimization modules are organized in order of target dependency. In the case of LLVM, the optimization modules are organized so that standardization optimization,e.g., dead code elimination, peephole optimization, loop invariant code motion, etc, comes first, followed by less-canonical or target-dependent optimization,e.g., aggressive inlining/loop unrolling, vectorization, and other target-specific optimization. Among standardization and target-dependent optimization, the problem of the order of various compiler optimizations, such as loop optimization, DCE, and peephole optimization, is an unsolved heuristic problem following inline expansion. Some research exists on its ordering by machine learning. [18] Vectorization, which is the focus of this paper, comes first among the target-dependent optimizations. A new target-independent IR MLIR [19] has also been proposed as a solution to the implementation variance and complexity of intermediate representations in various programming languages by allowing the representation of object lifetime and graph structures that LLVM IR could not represent. MLIR is a first-class concept of polyhedral loop optimization, which is easier to optimize specifically for parallel computation than existing intermediate representations and MLIR is also used for hardware description as an intermediate representation of high-level synthesis since it is designed in a way that allows the description of graph structures. [20] Recently, a language using MLIR, Mojo, has been developed, which is much faster than Python by utilizing SIMD instructions [3]. MLIR is excellent in terms of optimization performance in specific examples. Still, it has not yet replaced the backends of many generally used languages, so we target the more widespread LLVM IR in this paper.

2.4 Automatic Vectorization

Modern general-purpose processors have scalar operation instructions, which generally take 64/32-bit operands and produce 64/32-bit operands as core ISAs. A compiler optimization that converts a scalar program consisting of scalar operations into a vector program that uses vector extension instructions as described in the previous section, is called Vectorization, especially in the context of compiler optimization, and Automatic Vectorization in the context of compiler optimization. There are two main types of Automatic Vectorization algorithms: target-

independent and target-dependent. The latter is mostly just ad-hoc peephole optimization. The former target-independent and practical compiler automatic vectorization algorithm consists of Straight code vectorization; the most famous one is Superword-Level Parallelism Vectorization [21, 22], and Loop Vectorization components. For example, the former is called SLPVectorizer in LLVM, while the latter is called LoopVectorize. As described below, the distinction between them is not essential from an algorithm perspective, and some studies propose a unified Generalized SLP[22]. Also, target-independent IR, like LLVM IR, has vector-oriented peephole optimization done in the name of VectorCombine, like InstCombine, the general LLVM IR peephole optimization PASS. For existing research, VeGen[4], Vectorizer Generator inputs Developer manuals published from CPU-vendor, and produces ad-hoc peephole optimization codes handles LLVM IR with calculating the original cost model. Minoraur[5], SIMD-oriented Superoptimizer, searches ad-hoc peephole optimization patterns enumeratively using LLVM IR transformation validator alive2[6]. We tackled the Complementary searching of opportunities, especially between variable-length and fixed-length vectors, for target-independent optimization patterns and, in the future, target-specific optimization patterns.

2.4.1 SLPVectorizer

SLPVectorizer is also called a Straight block vectorizer. In contrast to Loop level optimizations, the SLPVectorizer is an automatic vectorization method for instruction levels. Like Figure 2.4 shows, it creates a tree of instructions and vectorizes the scalar instructions tree by grouping isomorphic operations on an instruction-by-instruction basis. Before vectorizing the scalar tree, it calculates the cost model to decide whether using vector instructions is profitable with the vector operand loading overhead tradeoff. This cost model calculation heuristic is improved year by year like various SLP variants [23, 24, 25]. Except for transformations of auxiliary structures such as loop unrolling, etc, the vectorizer is separated from LoopVectorize and performed on a basic block-by-block, instruction-level basis. The components are separated from LoopVectorize, described below, to simplify implementation, but cases where instruction-level vectorization can be performed across control structures are often overlooked. For example, since LoopUnrolling does not maximize vectorization opportunities but only looks at the tradeoff with binary size, it may miss SLPVectorization opportunities that LoopUnrolling could discover. LoopUnrolling may miss discoverable SLPVectorization opportunities. Research such as [26], which proposes a suitable LoopUnroll for SLPVectorizer purpose, and Generalized SLP[22], which offers an IR for finding vectorization opportunities on inter-control structure, have emerged in recent years.

2.4.2 LoopVectorizer

The Loop is the control structure with the most room for parallelization. Although a polyhedral model for loop optimization is proposed in the context of the Deep Learning compiler[27], the general compiler doesn't do polyhedral analysis by default but with more heuristic loop analysis and cost model calculations because of compilation time and complexity tradeoff. The vectorization of Loop is implemented as a separate component from the vectorization of other instructions, which is familiar to both GCC and LLVM. The LLVM implementation consists of the following three steps[28]

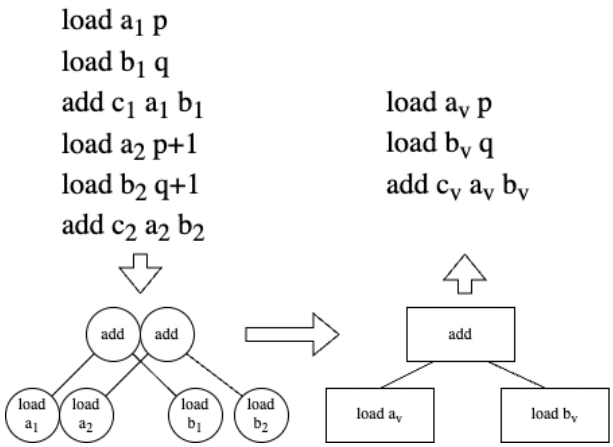

Figure 2.4: SLPVectorization

1. Legality check
2. Profitability check
3. Transformation

Legality check

Verify that the Loop can be rewritten by vector extension instructions, such as whether it uses induction variables, whether it performs calculations that are multiplied by the Reduction structure, whether memory access dependencies do not cross loops, and whether there are aliases in the case of multiple pointers.

Profitability check

Vector extension instructions should not be used whenever they are available. The execution of vector extension instructions is subject to restrictions on operand alignment, and even if the operation itself is parallel and effective, loading the values into the vector registers required to execute the operation is also expensive. We look at the cost model and variable alignment defined for each Target to estimate whether the vectorization of the loop operations will be efficient. The hardware vendor should manage this, but as a matter of fact, models managed directly by the vendor are not accurate either. It is also well known that the x86-64 cost model in LLVM is managed by Sony engineers[16]. As is also the case with SLPVectorizer, accurate cost model management is essential. Cost model management has also been studied using machine learning methods such as Ithemal[29].

Transformation

If the legality and profitability are guaranteed, the transformation is finally performed.

	Dead	Ubfuzz	Dfuser
Program Generation	CSmith	CSmith	CSmith
Program Mutation	Inserting marker macro For each basic block	Inserting UB 1) arithmetic operands, 2) array indexing, etc	1) Inserting code position pragma 2) Inserting optimization attributions 3)
Compared Compilers/ Profiles	1) -O2 vs -O3, 2) GCC vs LLVM 3) Clang/GCC version A vs B	-O0 vs -O3	-O3 -g vs -O3
Compared Relations	Inclusion of marker remainders	Whether optimized binary is crashed or not	Binary comparison

Figure 2.5: Comparison of prior Compiler Fuzzing methods

2.5 Compiler Fuzzing

2.5.1 Overview

In recent years, there has been a lot of research on automatic search for bugs and optimization opportunities in open-source compilers such as GCC and LLVM, using CSmith as a base for test generation to reduce the cost for compiler developers. DEAD, Dfusor, and UBfuzz[10, 11, 12] all find and report Issues by Fuzzing practical compilers. Dfusor[11] for automatic detection of binary mismatch with Debug compilations, UBfuzz for detection of False Negative (false alerts for programs that do not contain undefined behavior) bugs in Undefined Behavior Sanitizer, UBfuzz[12] for finding False Negative (false alerts for programs that do not contain undefined behavior) bugs in the Undefined Behavior Sanitizer, and DEAD Code Elimination DEAD[10], which performs optimization evaluation focusing on Dead Code Elimination. DEAD finds optimization regressions, Dfusor finds consistency between debug and release profiled builds, and UBfuzz finds false negative alerts in the sanitizer. All of these are reported differently, but the methods used should be the common framework applicable to other compiler toolchain management fields. This section will attempt to generalize and explain this common property.

Figure 2.5 is the Comparison matrix table for existing Compiler Fuzzing works, and more visually pictured by Figure 3.1 for vectorizer purposes, as this paper proposes. Roughly, the steps of Compiler Fuzzing can be divided into 1. Test Generation, 2. Difference Testing, and 3. Analysis. Test Generation comprises Generator and Mutator Difference Testing from the Compiler and Comparator and Analysis from the Reducer. Each component will be described as the following in comparison with related studies.

2.5.2 Components

Generator

Compiler Fuzzing needs to generate seed programs for differential testing. The first tool for this purpose is CSmith[7], a tool to generate C programs with various language features that do not include undefined behaviors. And other tools inspired by CSmith have been proposed recently[8, 9]. The actual generated program by CSmith is as follows.

...

```
int8_t ****l_962[10][8][3] = {{{&g_891,...}}};
int16_t l_1199 = 7L;
const union U0 *l_1240 = &g_247;
uint64_t ***l_1246 = &g_452;
int32_t l_1364 = (-1L);
uint32_t l_1408 = 4294967295UL;
int i, j, k;
for (i = 0; i < 9; i++)
{
    for (j = 0; j < 1; j++)
    {
        for (k = 0; k < 1; k++)
            l_598[i][j][k] = 0UL;
    }
}
for (i = 0; i < 2; i++)
    l_620[i] = &l_621[1][0][2];···
```

CSmith generated a randomly structured program like the one above, and the original purpose of CSmith is to use them as test cases for compilers to see if they do not crash or are incorrectly optimized by checking hash for global variables and return value. Many related studies use it as seed programs and transform them into programs with the respective properties of interest. To target LLVM-based language processing systems, which are often the target of vectorization and have various front ends and targets, we attempt to generate C programs using CSmith and then convert them to a form specialized for judging vectorization performance. As shown in Figure 2.5, many related studies use CSmith. We believe this is because CSmith is easy to use since it generates programs that do not contain Undefined Behavior. Dfusor uses yet other random C program generators tkfuzz[30] and Hermes[31].

Mutator

Programs generated with CSmith can optionally be given restrictions on the language features used by the program they generate, such as whether to include volatile variables (−−no−volatiles), whether to take standard input (−−no−argc), and so on. However, we do not know whether they have the necessary properties for the bugs or opportunities we wish to find. Therefore, for efficient and effective finding, it is necessary to transform the generated programs into programs with the properties of interest. For example, in the case of a program like UBfuzz[12] that is specialized for Sanitizer bugs, CSmith is guaranteed to generate a program that does not contain any undefined behavior, so the following macro can be inserted and used to cause an out-of-domain reference or signed integer overflow in array accesses.

```
int32_t MUT_VAR = +(2147483645)/*UBFUZZ*/;
#define _INTOPR0 +(MUT_VAR)
#define _INTOPR1
...

  a = ((((a++) _INTOPL1)
      + ((1) _INTOPL0)) _INTOPR1)
      + ((++c) _INTOPR0);
...
```

In the method proposed in this paper, we use the Clang AST Transformer API as in the related work. By using that, we transform the C program generated by CSmith into 1) the program has the array alignment to 128 bytes alignment and the size of 4096 for each dimension, and 2) has sufficiently bigger size of loops in which the induced variables of the program are incremented. As we will see later in the evaluation section, this sufficiently increases the vectorization opportunities of LoopVectorize but has little effect on the SLPVectorizer. As we may have learned from the background description, it would be better to prepare a set of pre-packable isomorphic instructions at points in the various control structures and insert them for the vectorization opportunities targeted at the SLPVectorizer. Preparing and inserting pre-packable instructions at various points in the control structure would be a good idea.

Compiler Driver

After generating a program containing the properties to be verified by Mutator, compile the program to compilers with various profiles to be compared with other profiles. The target of the compilation depends on the property to be verified and the structure of the compiler's intermediate representation. In the case of dead[10], the compilers are GCC and Clang, each with different versions and optimization levels for the same compiler. In the case of dead, the comparator described below uses macros, so regressions can be found because the compilation results can be compared using dead code marker metrics on Dead Code Elimination. Dfusor[11] compiles with the same optimization level and whether the debug information is valid to see the Compilation Consistency Modulo Debug Information, which shows no binary difference of binary on debug build from release build, and the UBfuzz compiles without optimization and with optimization to see the false negative alert of undefined behavior sanitizer. Having representations in which Reduction can be easily performed is also desirable.

Comparator

Compare the compilation results so that the properties you want to detect are known. In the case of DEAD, we compare different compilation results focusing on the program's control structure to see if the inserted macro has disappeared. We can compare programs by total order, and we can find the regression regarding dead code elimination. Dfusor compares instructions and finds the difference between release and debug build. Not only program comparison but also dynamic analysis of the program would work to see the program characteristics; UBfuzz uses debug information to see if the undefined behavior happens in the exact location.

Reducer

As we see in the Generator and Mutator section, the differential test program isn't good for humans to understand or debug based on that. First, CSmith produces massive redundancy and unnecessary complexity, so we must reduce it. CReduce [32] is the popular tool for test reduction for various languages. Since the goal of Compiler Fuzzing[10] is to find regressions related to optimization for compiler development, undefined behavior might be a false alert, which can be optimized away in any way if the instruction is executed or the value is used depending on the semantics of the language. To prevent that DEAD uses a program verifier CompCert[33] to see if undefined behavior is contained in the tests when regression candidates are found. Also, DEAD can bisect the repository commit log to identify the commit from which the regression originated and so on. Such engineering is also important since the main purpose of this field is to improve the ease of debugging in actual compiler development.

Chapter 3

Vectorizer Fuzzer for Finding Missing Vectorization Opportunities

In this chapter, we describe our proposal of Fuzzing for Compiler Vectorizer. As mentioned in the Introduction, to achieve fully automated compiler fuzzing, we should design the following three elements

1. Test Generation
2. Difference Testing
3. Analysis

We propose and implement a novel method for Test Generation and Difference Testing that is effective for target-independent Automatic Vectorization algorithms, especially for LoopVectorize. First, an overview of the proposed method is given, followed by a description of each Component corresponding to the Framework of Compiler Fuzzing described in the previous chapter.

3.1 Overview

Figure 3.1 is an overview of the proposed Vectorizer Fuzzer. Briefly saying each component works as follows.

1. Generator Test program generation with CSmith.
2. Mutator: Mutate the program generated by CSmith for Vectorizer Differential Test (VecMut).
3. Compiler Driver: Compile the mutated test program with different target information or different compilers.
4. Comparator: Make the compiled results comparable and compare them.
5. Reducer: Reduce the original program when a test case is found.

Two points of particular interest are as follows

- Test Mutation using Clang AST to introduce Automatic Vectorization opportunities
- Comparator for finding the quantitative difference of two Vectorization

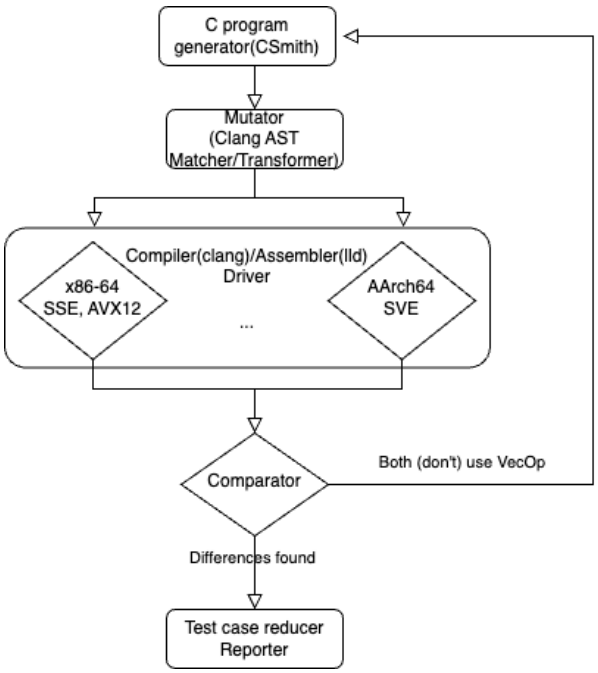

Figure 3.1: Overview of proposal Vectorizer Fuzzing

We use C-compiler Clang for the compiler frontend, which is based on universal compiler infrastructure LLVM, and Generator and Reducer are based on CSmith/CReduce, respectively. We do not make any restrictions on the language functions for CSmith generation, so that we can see various patterns for vectorization. The Legality Check of Vectorization should also determine whether aliases are present and whether or not they are volatile, so there are no restrictions on pointers, and various program language features CSmith supports.

3.2 VecFuzz Components

3.2.1 Mutator: Loop-oriented Vectorize opportunities injection

First of all, in conclusion, we implemented the following three types of transformations for injecting vectorization opportunities for programs generated by CSmith.

1. Enforce the declaration alignment of any dimensional array to 128-byte alignment.

2. Set the size of the array to 4096.

3. Change incremental loop bound to 4096.

They are mainly motivated to introduce beneficial loops for vectorization on any targets.

First, to know the rationale above transformations, let us observe the loop program generated by CSmith. Listing 3.1 is the part of the program generated by CSmith that contains the loop.

Listing 3.1: Loop Program generated by CSmith[7]

```
...
/* ---- Struct/Union Declarations ---- */
/* ---- GLOBAL VARIABLES ---- */
static const uint64_t g_8[5] = {1UL,...};
static int32_t g_10 = 0x8CE3E63EL;
static int32_t g_11 = 0L;
...
    int16_t l_3217[9][3][5] = {{1L,...};
    int16_t l_3231 = 0xAC06L;
    const uint32_t l_3268 = 0xDB7395C5L;
    int i, j, k;
    for (i = 0; i < 6; i++)
        l_3155[i] = 0x5357627B9C38BE8CLL;
    for (i = 0; i < 1; i++)
        l_3216[i] = 4294967295UL;
...
```

There are a large number of loop structures used for embedding in the code generated by CSmith. However, since the size of the loop bounds and arrays are very small, the compiler's cost model determines that the overhead of loading into the vector register is dominant, and these loops are not worth it to be vectorized.

Discussing the cost model is also essential for vectorization quality. However, we want to ignore the debate on the cost model in this article to focus on legality checking joint to various hardware targets for complementary searching of vectorization opportunities. Let us summarize the properties of an ideal test program for complementary searching of opportunities for Vectorizer this time. The following are the main essential aspects of tests generated by Fuzzing.

- The structure must contain various kinds of code snippets using interesting features to be found that can be observed without being affected by other optimizations, etc.
- The root cause of the bug or property to be found, and information that identifies the location of the bug or property.

Regarding the first point, since the alignment of each operand, the number of loads for the target operation, and the trip counts of every loop for the vector register size are such that cost models are involved, we thought it necessary first to convert the program to handle sufficiently large, properly aligned data. To make the loop vectorizing profitable, we make the loop trip counts bigger enough. Also, realignment is out of the discussion here; we should profitably align the declaration of the loop using the alignas specifier introduced in the C23 standard. In addition, LoopVectorize generally uses a scalar reminder loop, known as an Epilogue loop, to handle the loop steps that are to be divided by the size of the vector register. Vectorization algorithm, but how the epilogue is handled is also important for the Loop Vectorization algorithm. Loop Vectorizing for the loops with odd trip counts is ad-hoc. Without predicate or masking registers like Arm SVE or x86-64 AVX-512, we need to process the remainder of the operand size of vector instructions by the Scalar loop. That criterion is target-dependent and not transferable for each target. The boundary between the Vectorized loop and the Scalar Loop process is calculated using the Loop

Invariant and other heuristic methods, so we designed the loop size to be the power of two larger and fully vectorizable to find the transferable vectorization opportunities. Based on the above discussion, we decided to change the size and alignment of the array using the Clang AST Transformer for this evaluation. We set the alignment to 128 bytes and the size to 4096. Regarding the second point, we have not been able to consider it in this evaluation because we used statistics computed inside the compiler for preliminary purposes, which are location and root cause independent. However, in the case of automatic analysis, it would be a good idea to use debugging information, as UBfuzz did, to consider 1) whether or not vectorization has occurred and 2) where the vectorization occurred in the source code. In addition, mutations to programs with isomorphic instruction-level opportunities are future work. We believe inserting multiple operations of the same type is a good idea for basic blocks of various control structures.

Implementation

Based on the above discussion, we decided to change the C program generated by CSmith using the Clang AST Transformer for this evaluation. We mutate the alignment to 128 bytes and the size to 4096 automatically. Here is a brief description of how the Clang AST Transformer can be used to write Matcher and Replace rules. Listing 3.2 is a Clang AST Matcher that matches the declaration of an N-dimensional array.

Listing 3.2: Clang AST Matcher for N-dimension Array Declarations

```
auto ArrayMatcher =
    constantArrayType(
      hasElementType(unless(arrayType()))
    ).bind("arrayType");
for (unsigned i = 1; i < n; ++i) {
  ArrayMatcher = constantArrayType(
    hasElementType(ArrayMatcher)
  );
}
return varDecl(
  hasType(ArrayMatcher),
  optionally(
    hasInitializer(
      initListExpr().bind("arrayInitializer")
    )
  )
).bind("nDimArray");
```

We can write a functional way to bind the AST-based matches in the above Matcher as string variables and replace them with text using them. 3.3 is a Rule that changes the alignment to 128 bytes and the size to 4096 for array declarations.

Listing 3.3: Replacement Rule for Array Declarations

```
std::string NewSize = "";
for (unsigned i = 0; i < n; i++)
  NewSize += "[4096]";
return makeRule(
    makeNDimensionalArrayMatcher(n),
    flatten(ifBound(
        "arrayInitializer",
        changeTo(
          cat("alignas(128) ",
            between(
              before(
                node("nDimArray")
              ),
              name("nDimArray")
            ),
            name("nDimArray"), NewSize, "=",
            node("arrayInitializer"), ";")),
        changeTo(cat(
            "alignas(128) ",
            between(
              before(
                node("nDimArray")
              ),
              name("nDimArray")
            ),
            name("nDimArray"), NewSize, ";"
            )))));
```

Similarly, by defining and applying a Rule that changes the bounds of the loop to 4096, all declarations and loops are mutated. For example, 3.1 is transformed to 3.4.

Listing 3.4: 3.4 mutated by VecMut

```
alignas(128) int16_t *l_1032[4096];
alignas(128) uint16_t l_1034[4096];
alignas(128) uint32_t l_1043[4096];
int i, j, k;
for (i = 0; i < 4096; i++)
    l_1032[i] = (void*)0;
for (i = 0; i < 4096; i++)
    l_1034[i] = 0UL;
for (i = 0; i < 4096; i++)
    l_1043[i] = 1UL;
```

3.2.2 Comparator: Inter-Target Vectorizer Comparator

In conclusion, in this complementary Vectorizer Fuzzing Comparator, we use LLVM's statistical tools for compiler optimization developers, especially for the following three numbers. They are 1) the number of loops analyzed by LoopVectorize, 2) the number of loops vectorized by LoopVectorize, and 3) the number of instructions vectorized by SLPVectorizer. 1) is the number LoopVectorize mod-

ule trying to check Legality and Profitability; it's affected iteratively whether the loop is vectorized or unrolled. And 2 and 3 are the number of objective structures for each vectorizer. We compare one of those corresponding values for comparisons for which vectorizer we want to find the missing opportunities. Listing 4.1 is part of the output of the compile time optimization statistics for developers for the program in listing 3.4.

In general, for Compiler Fuzzing, it would be ideal if the Comparator could drop in metrics from assembly or binary with the following properties.

1. Two program could be quantitatively compared in interesting characteristics

2. The value is proportional to the missing opportunities/bugs caused by implementations, not the unique target specifications.

Regarding the first point, our statistic comparison satisfies. We can compare target-independent vectorizer performance for each target quantitatively. However, the above optimization statistic method does not fully satisfy the latter property mentioned earlier. Since what we are looking at in this experiment is only the number of loops and instructions checked by the Vectorizer, we can see which of the two compilations can Vectorize more. However, what is important when fixing bugs is whether the difference between the two compilations is due to the specification of the target vector extension instructions or simply an omission in the implementation. I think it is a sound algorithm because when there is an actual implementation difference, there is also a statistical difference, but designing the metrics more into ISAs for each target would make the tool more useful for compiler developers. More ideas will be discussed in Future work. DEAD[10] is the same; found opportunities are not always tractable. For vectorizer cases, this requires a lot of knowledge about comparing architectures, what is common to them, and what is not. For our evaluations, to reduce these kinds of differences caused by specifications, make the hardware specs the same as well as possible, e.g., machine register size, numbers, and cache line sizes, while keeping the interesting differences as it is, e.g., variable or fixed size vector registers. Also, we can cover these subtle missed opportunities by reducing the program while keeping the same characteristics as the original one.

Chapter 4

Evaluation

In this chapter, we describe the experiments we conducted to evaluate the effectiveness of the Vectorizer Fuzzer. We will experimentally show Test Generation and Differential Testing are realized regarding the Vectorizer Fuzzer, out of the three compiler fuzzing elements, as shown in the introduction. The two objectives of the evaluation at this stage are

1. CSmith + Vectorizer Mutator can generate appropriate tests for the difference test of Automatic Vectorization.

2. we can observe the difference in Vectorization between AArch64 SVE and x86-64 AVX-512 compilations.

Corresponding to each objective, the following two types of experiments were conducted. The details of the settings, etc., are described in the theory of each evaluation.

1. Two programs, the C program generated by CSmith and its mutated version by Mutator for Vectorizer Fuzzing, were compiled for a model with AVX-512 on Intel x86-64. Compare the number of optimizations of LoopVectorize and SLPVectorizer.

2. Two programs, the C program generated by CSmith and its mutated version by Mutator for Vectorizer Fuzzing, were compiled for the model with Armv8 SVE and the model with AVX-512 on Intel x86-64. Compare the obtained numbers with the compiled results.

The first experiment, this kind of experiment is also done to show the effectiveness of test generation on Dfusor[11]. Also, for the second experiment, we benchmark the TSVC for the Arm SVE and x64 AVX-512 and show those targets' original potential for vectorizing.

4.1 Effectiveness to introduce vectorization opportunities

In this section, we will check whether the VecMut described in Chapter 3 increases the opportunity of the Vectorizer.

4.1.1 Experimental Settings

For this evaluation, we will use the stats of SLPVectorizer and LoopVectorize, target-independent vectorizers provided by the developer of LLVM. 1) Compile the program generated by CSmith as it is, 2) Apply VecMut to the program generated by CSmith and compile it, and compare the Stats of the two compilations.

For example, Listing 4.1 is a part of Stats compiled by CSmith. We can see various optimization stats for each optimization module in LLVM.

Listing 4.1: LLVM Statistics Example

```
...
  308 aa              - Number of MustAlias results
15782 aa              - Number of NoAlias results
    3 argpromotion    - Number of dead pointer...
  446 assume-queries  - Number of Queries ...
   13 loop-vectorize  - Number of loops analyzed...
    4 loop-vectorize  - Number of loops vectorized
...
```

The target is x86-64 avx512, using the option ignoring the SLPVectorizer cost model to omit the cost model discussions. This experiment relies on LLVM because of the use of LLVM internal statistics. Still, it could also be verified for GCC, or compilers developed by Intel or Fujitsu if there were compiler-independent, black-box vectorization metrics that could be quantitatively known, but that search is future work.

4.1.2 Experimental Environment

We experimented with Ubuntu 20.04 running on Intel Xeon Gold 6326 16c/32t x2, 256GB, Optane Memory 1024GB. The version of LLVM Clang under test is 18.0.0 (5d59e97e88), with assertion disabled and statistics enabled release build. The version of CSmith was 2.4.0 (92069e4).

Implementation

The proposed method was implemented in C++. Among the implementations, VecMut was implemented using the Clang AST Transformer API. Also, VecMut performs transformations as described in Chapter 3.

4.1.3 Number of Vectorized/Analyzed Loops and Vectorized Instructions

The programs generated by CSmith 10,000 times are mutated by VecMut and compiled with Clang 18.0.0 with the stats option enabled. Its overall execution took about eight hours.

Figures 4.3, 4.1, and 4.2 are histograms representing the results of this experiment. The horizontal axis is the value of stats, the vertical axis is the number of test cases that have corresponding values, and the dashed line in the figure represents the corresponding colors' average. Blue is the corresponding stats value when compiled using CSmith as-is, and red is the corresponding stats value when VecMut is applied. Figures 4.1 and 4.2 show that VecMut increased the number of loops to analyze by an average of 67% and the number of loops to vectorize by an average of 60 % increase in the number of loops to vectorize. Although we cannot analyze the details of this compilation because it is −O3 and is affected by other LoopUnrolling and LoopInterchange optimizations, The fact that the number of Analyzed loops has increased indicates that VecMut increases the number of vectorization opportunities or suspicious loop structures. Namely, this means that the number of loops that the LoopVectorizer determines if it can vectorize using the Legality and Profitability Check has increased, The vectorized loops number also increased, which might have increased the analyzed numbers

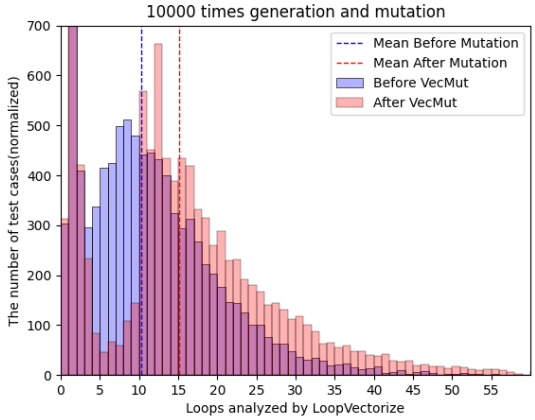

Figure 4.1: Number of loops to analyze in LoopVectorize: After VecMut - red, CSmith - blue

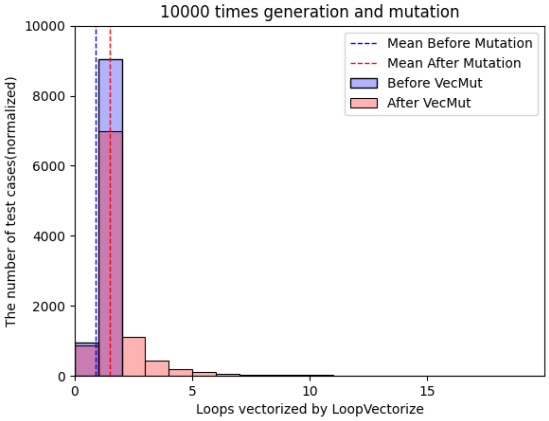

Figure 4.2: Number of loops to vectorize in LoopVectorize: After VecMut - red, CSmith - blue

in the chain. This means that the goal of this evaluation, "CSmith + Vectorizer Mutator can generate appropriate tests for the difference test of Automatic Vectorization," was achieved when LoopVectorize was used as the target. However, for the SLPVectorizer, it is impossible to generate appropriate tests. There is almost no change between using the CSmith test program and applying VecMut, this is since, this time, only Mutation was focusing on the Loop structure, not isomorphic instructions. To achieve finding complementary SLPVectorization opportunities we need to further consider what kind of instructions should be vectorized common to another target, this would be future work.

4.2 Effectiveness to find vectorization difference for each target

The above Test Generation strategy effectively shows Test Generation for the vectorizer, but we aim to find the bugs/opportunities in the vectorizer. In that case, the experiment conducted here is to see whether we can find the inter-targets missing vectorization opportunities in Differential Tests between different compilation targets. As evaluation targets, we selected ArmV5 SVE, which has variable-length and fixed-length vector registers, and x86-64 AVX-512, which has a fixed-length vector register. In particular, we wanted to focus on variable-length

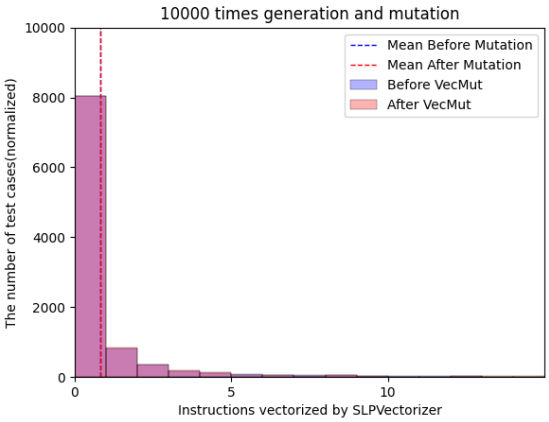

Figure 4.3: Number of instructions to vectorize in SLPVectorizer: After VecMut - red, CSmith - blue

Table 4.1: SVE and X86 TargetTransformInfo on Evaluation

Target	Fixed Vector Register Width	Scalable Vector Register Width	MinVector Register Width
x86-64 AVX512	512	0	128
AArch64 SVE	512	128	64

vs. fixed-length vectorization. This is because variable-length vectors require tricky handling in Vectorizer and even in verifiers such as Sanitizer, alive2[6], etc., and many implementations are incomplete. However, inherently, variable-length vector registers should have more optimization opportunities than targets with fixed-length vector registers. So, this time's objective is to find the missing vectorization opportunities between fixed-length and variable-length vectorization complementary opportunities.

Also important here is to figure out if the hardware model properties of the targets being compared are relatively close or if there is a dominant relationship between one or the other. In truth, we need to ensure that ISA's semantics are fully inclusive; that would be another work in the future. Still, here, we configure the abstraction, TargetTransformInfo, which represents the target-independent values used in LLVM's optimization path, the same value as possible as we can for the values that LoopVectorize and SLPVectorizer refer to. The table of target hardware values in this evaluation is shown in Table 4.1. We might know in this setting that the target, Arm with SVE, should be more powerful than the target, x64 with AVX-512, regarding the Vectorization-related hardware model conceptually, but it should be better to evaluate it experimentally for major benchmarks.

4.2.1 TSVC Benchmark for x86-64 with AVX512 and AArch64 with SVE

For the general benchmark of automatic vectorizing compilers [34, 35], we plotted the vectorizer statistics when compiling/cross-compiling with LLVM Clang version 18.0.0 (5d59e97e88) using Ubuntu 20.04 running on Intel Xeon Gold 6326 16c/32t x2, 256GB, Optane Memory 1024GB, same to the experiments in the previous section. The commands used for AVX512 are the same as in

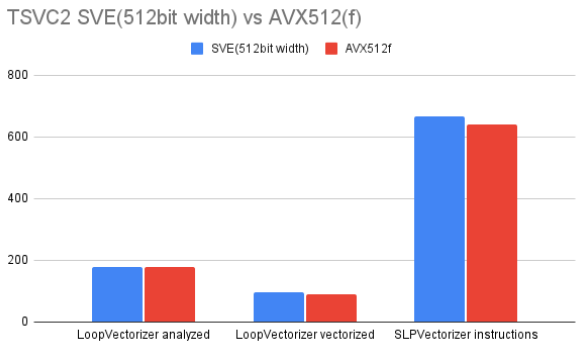

Figure 4.4: Vectorizer related stats for TSVC benchmark for Intel x86 with AVX-512 and ArmV8.5 with SVE

the previous section, and the commands used for Arm64 SVE changed the target and fixed vector register size by −target aarch64, −march=armv8.5−a+sve −msve−vector−bits=512

Figure 4.4 is the result. In the TSVC2 benchmark, the Vectorizer performs better on the SVE with variable-length vector registers, which seems rational from the hardware register size settings from Table 4.1.

For a famous benchmark like TSVC2, it seems naturally well implemented, and the number of Vectorized SVEs can take a greater variety of vector lane sizes. The next section will examine the differences in CSmith-based programs between SVE and AVX-512.

4.2.2 Number of found vectorizer differences

As in the previous section, the programs generated by CSmith 10,000 times are transformed by VecMut and compiled with Clang 18.0.0 with the stats option enabled. Compare the compiled stats for Intel x86 with AVX-512 and ArmV8.5 with SVE. Note that Armv8 is cross-compiled on x86 machines.

The sum of the differences in the statistics related to the Vectorizer found is summarized in 4.2. First, we can consider that the SVE target contains actual bugs and opportunities since there are many situations where the x86 side excels as stats against the variable length concepts and a previous experiment. In contrast, the x86 target may have fewer vectorization opportunities than the Arm target. As for SLPVectorizer, it is difficult to know if it is a bug or not without reduction because of the large instruction dependence and the complexity of the cost model calculation. In the same way, the results when a file like the one generated by CSmith is given as input as is are also plotted4.3. As we saw in the experiments in the previous section, for LoopVectorize, there are clear differences that can only be found after applying VecMut. This could also be evidence for the effective mutator. As for SLPVectorizer, differences hidden by VecMut and differences found by VecMut are 78 and 77, respectively. Therefore, it cannot be said that the VecMut implemented in this study generated effective code for verifying the SLPVectorizer.

Table 4.2: Number of Loops/Instructions that LoopVectorize/SLPVectorizer vectorized more out of 10000 trials of VecMut after CSmith on Intel x86 with AVX-512/ArmV8.5 with SVE

Target	LV vectorized	SLP instructions
x86-64 AVX512 +	9	92
AArch64 SVE +	15	201

Table 4.3: Number of Loops/Instructions that LoopVectorize/SLPVectorizer vectorized more than Intel x86 with AVX-512/ArmV8.5 with SVE out of 10000 trials using CSmith as is

Target	LV vectorized	SLP instructions
x86-64 AVX512 +	0	90
AArch64 SVE +	0	204

4.3 Discussions and Future Work

Analysis and Reduction for the difference

The numbers may be mostly structure-specific, introduced by the Mutator this time, e.g., bigger alignment and 4096-sized loops and arrays. So, the number seen here might have been attributed to just one bug or missed opportunity. To avoid this, we can randomly set the loop bounds and array size big enough to ignore the cost model because that's one room for the opportunities. Also, the automatic reduction is necessary to analyze the above program to search in the complicated CSmith-based program for the root cause of those differences. This should be possible if we do CReduce[32] in the same way as DEAD[10] while comparing the size of the statistics found, but this has not yet been done here. CReduce allows users to describe an interesting test and bisects the program with various heuristics while ensuring the program passes the test. Although the fact that differences exist is intriguing, we still need to do more analysis about that.

Backend Ad-hoc Peephole Optimization Opportunities

This time, the Comparator's design aimed to detect the difference in Vectorization that occurred in the target-independent middle-end. Still, since it is middle-end stats, it is naturally less effective for the backend part of the Vectorizer implementation, especially target-dependent Ad-hoc peephole optimization pattern caused by cost model and instruction kinds. In particular, to search for ISA-specific peephole optimization patterns in Target, it would be better to use the semantics of vector extensions support system Z3[36] or its applications, which handles axiomatically for ISA instructions, such as Minotaur[5] and VeGen[4] does. Inherently, target-independent missed opportunities and target-dependent missed opportunities might be better handled separately since mixing them would be too complicated and blurry for feedback. Lane Level Parallelism proposed by VeGen[4] is particularly interesting. It would be good to consider using LLP for Comparison on Differential Testing. Also, it can be used to extend the current target-independent algorithm, SLPVectorizer implementation, cause LLP is proposed as an extension to SLP.

Random twiddling of control structures, optimization

The Mutation on this Fuzzing focused on the middle-end of the compiler loop structures this time and experimented with a single preset optimization option −O3. However, the current middle-end optimization is generally a very complex and black-box approach, as it depends on the order of optimization, like the optimization performance problem as known as PhaseOrdering[18] in the LLVM community. We believe that a method such as Twiddling optimization attributes like DEAD[10], Dfusor[11] randomly would be effective in this Vectorizer-oriented fuzzing as well. DEAD changes the options for CSmith test generation depending on the random number. Dfusor adds various optimization attributes (Fine control optimization) and transforms the control structure (Random Code Lowering) in Test Generation, the Mutator phase on Compiler Fuzzing. In particular, we believe that general mutations related to control structures, such as obfuscation techniques, are adequate for the Vectorizer Fuzzer. Although only Stats was used for comparison this time, if the focus is on the middle-end, a more structural comparison could have been made using LLVM IR and its Control Flow Graph(CFG) structure with debugging information added or IR of alive2 from which conversions can be made. LoopVectorize can change the program's control structure significantly when optimization is performed, such as when both Scalar and Vector loops are used. A Comparator focusing on the structure of CFGs in LLVM IR might also be practical if implemented.

Cost Model Fuzzing, Dynamic Comparison

As mentioned repeatedly in the proposed method, one of the design policies for fuzzing is to ignore the cost model and focus on finding the common optimization opportunities between targets. As a command, it is implemented to invalidate the cost model calculated by SLPVectorizer. So, the missed opportunities attributed to the cost model are harder to find on this Fuzzing. If Reduction works normally, it will be easier for compiler developers who are familiar with a target to analyze the cost model afterward, and thus, with the above-mentioned minor innovations, Fuzzing like this may be used as Fuzzing about the cost model. In addition, the cost model of the current ISA is often done through experiments by compiler developers, and the testing method itself is left to the hands of individual developers, so there is a constant debate about the validity of the cost model. In particular, the metrics used in the Comparator need not be limited to those that can be statically obtained at compile time. Even without an actual device, a simulator such as qemu[37] may be able to automate the management of cost models with a certain degree of accuracy measured by developers' machines. Also, it is possible to dynamically analyze the frequency of use of vector registers, etc., so it may be possible to perform more accurate reduction and analysis. Dynamic analysis may be a good idea to try.

Chapter 5

Related Work

5.1 Random Program Generator and Differential Test

CSmith[7], used in this study, is a tool for generating random C programs. CSmith is proposed as a differential testing tool by hashing global states and return values to see if the compiler crashes or mis-compiles. Of course, the range to be tested by CSmith is limited, and optimization performances cannot be measured by CSmith itself, so many Differential Testing variants with Mutator are proposed as we see in this paper repeatedly[10, 11, 12].

Following this example, compiler test generators for other languages have recently been proposed in various domains. For example, NNSmith[9] for deep learning compilers generates random complicated ONNX neural network models to do fuzzing DL compilers, and RustSmith[8] is used for Rust compiler.

5.2 Vectorizer Generator

Lane Level parallelism (LLP), a more generalized version of SLP[21, 22] characterized by uniformity of operations and high operand, is proposed by Chen et al. [4]. VeGen, which uses LLP internally, automatically generates code for ad-hoc peephole optimization on the backend of LLVM based on the vendor's publicly available developer manual. This tool could be a powerful tool to manage target-dependent vectorizations, while our approach mainly focuses on target-independent vectorizations in this paper. We could extend our fuzzing approach by using this abstraction to find target-independent missing opportunities.

5.3 Enhancing Automatic Vectorization algorithm

5.3.1 Extended SLPVectorizer

This paper approached vectorizer performance improvements by lowering the costs to manage and find vectorization opportunities while much research tries to improve the qualities of vectorization algorithms. SLP[21, 38] is proposed and is often used together with LoopVectorize and managed separately. Integrated Vectorizers are proposed. One is SLP[22], which integrates them, has also been proposed in recent years, and another is Loop Unrolling unifying approach[26]. Vectorize management is complex because of inherent target dependency, as seen in this paper. Therefore, black-box approaches like this paper would be valued in practical compiler development; simultaneously, vectorization becomes powerful and versatile.

Instruction Cost Model management

One of the extensive management task-related vectorizers, Compiler Fuzzing, might be a variant solution, as is Instructions cost model management. IThemal [29] proposed the approach using machine learning and claims the effectiveness of that.

5.4 Superoptimizer

Superoptimizer has also been studied recently in the context of automatic vectorization, while it was initially claimed for general optimizations[39, 40]. Z3[36] is the theorem solver used in them. The two other solvers, souper[40] and minotaur[5] are LLVM-based super optimizers. The former handles peephole optimization, and the latter handles vector peephole optimization using LLVM-level intrinsics. As for minotaur[5], they use alive2[6] to validate the transformation candidates and calculate the profit by LLVM Machine Code Analyzer, llvm-mca of the program generated in an enumerative way vector augmented Intrinsics. Minotaur searches peephole optimization opportunities using vector intrinsics, but it takes much time to compile because of its enumerative verification/computation costs. Also, optimization using reinforcement learning[41] or other machine learning approaches[42, 43] are proposed. They can be understood as superoptimizers using machine-learning heuristics.

Chapter 6

Conclusion

This study proposes a method of automatic bug search for compiler optimization, especially for the automatic management of Vectorizer, which is highly target-dependent and has high complexity. Looking back at the following three elements of Compiler Fuzzing raised in the Introduction.

1. Test generation
2. Difference testing
3. Analysis

Of the three components, we believe that we have shown that the test generation and difference testing are effective against LoopVectorize, which is target-independent automatic vectorization, although it is limited to LLVM. In addition, the implemented Mutator is very simple, and we believe we were able to demonstrate its extensibility.

Since test generation and differential testing are not fundamentally independent, it is important to have a consistent design. If the method is more formal, defining the semantics of vector extension instructions like VeGen, and then using equivalence judgments, it would be good to port the optimization pattern of vector extension instructions of one target to another target. Designing a Comparator using such a method is a future work. We believe that the relatively simple implementation of Mutator and Comparator in this study is sufficient to demonstrate the possibility of automatic management of compilers, However, the theme of solving the difficulty of managing complex compiler optimizations, including automatic vectorization, has not been fully explored. The framework used for Fuzzing applies not only to Vectorizer, but also to various compiler issues such as mis-compilation, performance regression, and compile crashes, and the design of Mutator and Comparator specialized for these issues can be applied. We believe it is worth experimenting with various heuristics concerning We hope this research has contributed in some small way to lowering the cost of compiler development worldwide.

References

[1] "Intel® advanced vector extensions 10," July 2023. [Online]. Available: https://cdrdv2-public.intel.com/784267/355989-intel-avx10-spec.pdf

[2] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, p. 26–39, Mar. 2017. [Online]. Available: http://dx.doi.org/10.1109/MM.2017.35

[3] "How mojo gets a 35,000x speedup over python," December 2023. [Online]. Available: https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1

[4] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "Vegen: A vectorizer generator for simd and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 902–914. [Online]. Available: https://doi.org/10.1145/3445814.3446692

[5] Z. Liu, S. Mada, and J. Regehr, "Minotaur: A simd-oriented synthesizing superoptimizer," 2023.

[6] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for llvm," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 65–79. [Online]. Available: https://doi.org/10.1145/3453483.3454030

[7] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, jun 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[8] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1483–1486. [Online]. Available: https://doi.org/10.1145/3597926.3604919

[9] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association

for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[10] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 697–709. [Online]. Available: https://doi.org/10.1145/3503222.3507764

[11] T. L. Wang, Y. Tian, Y. Dong, Z. Xu, and C. 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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 146–158. [Online]. Available: https://doi.org/10.1145/3575693.3575740

[12] S.-H. L. et al., "Ubfuzz: Uncovering undefined behavior in c programs for security," 2023. [Online]. Available: https://shao-hua-li.github.io/files/2024_ASPLOS_UBFUZZ.pdf

[13] "x86/x64 simd instruction list (sse to avx512)," December 2023. [Online]. Available: https://www.officedaytime.com/simd512e/

[14] "The llvm compiler infrastructure," December 2023. [Online]. Available: https://llvm.org/

[15] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[16] "Cgo 2022 keynote: Compiler 2.0 by saman amarasinghe," December 2023. [Online]. Available: https://www.youtube.com/results?search_query=compiler+2.0

[17] "Global instruction selection," December 2023. [Online]. Available: https://llvm.org/docs/GlobalISel/index.html

[18] T. Jayatilaka, H. Ueno, G. Georgakoudis, E. Park, and J. Doerfert, "Towards compile-time-reducing compiler optimization selection via machine learning," in *50th International Conference on Parallel Processing Workshop*, ser. ICPP Workshops '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458744.3473355

[19] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," 2020.

[20] "Cglobal instruction selectionirct: Lifting hardware development out of the 20th century " by andrew lenharth, and chris lattner at llvm developer meeting 2021," December 2023. [Online]. Available: https://www.youtube.com/watch?v=ee01_yHjs9k

[21] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *SIGPLAN Not.*, vol. 35, no. 5, p. 145–156, may 2000. [Online]. Available: https://doi.org/10.1145/358438.349320

[22] Y. Chen, C. Mendis, and S. Amarasinghe, "All you need is superword-level parallelism: Systematic control-flow vectorization with slp," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 301–315. [Online]. Available: https://doi.org/10.1145/3519939.3523701

[23] V. Porpodas and T. M. Jones, "Throttling automatic vectorization: When less is more," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 432–444.

[24] V. Porpodas, A. Magni, and T. M. Jones, "Pslp: Padded slp automatic vectorization," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. USA: IEEE Computer Society, 2015, p. 190–201.

[25] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes, and T. Mattson, "Super-node slp: Optimized vectorization for code sequences containing operators and their inverse elements," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 206–216.

[26] R. C. O. Rocha, V. Porpodas, P. Petoumenos, L. F. W. Góes, Z. Wang, M. Cole, and H. Leather, "Vectorization-aware loop unrolling with seed forwarding," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3377555.3377890

[27] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, p. 708–727, Mar. 2021. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2020.3030548

[28] "2013 llvm developers' meeting: "vectorization in llvm"," December 2023. [Online]. Available: https://youtu.be/TVV5v5R43nA?feature=shared

[29] C. Mendis, A. Renda, D. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 4505–4515. [Online]. Available: https://proceedings.mlr.press/v97/mendis19a.html

[30] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 294–305. [Online]. Available: https://doi.org/10.1145/2931037.2931074

[31] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," *SIGPLAN Not.*, vol. 51, no. 10, p. 849–863, oct 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984038

[32] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," *SIGPLAN Not.*, vol. 47, no. 6, p. 335–346, jun 2012. [Online]. Available: https://doi.org/10.1145/2345156.2254104

[33] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Pister, and C. Ferdinand, "Compcert - a formally verified optimizing compiler," 01 2016.

[34] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '88. Washington, DC, USA: IEEE Computer Society Press, 1988, p. 98–105.

[35] "Tsvc2," December 2023. [Online]. Available: https://github.com/UoB-HPC/TSVC_2

[36] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[37] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[38] I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware slp in gcc," pp. 131–142, 01 2007.

[39] H. Massalin, "Superoptimizer: A look at the smallest program," *SIGARCH Comput. Archit. News*, vol. 15, no. 5, p. 122–126, oct 1987. [Online]. Available: https://doi.org/10.1145/36177.36194

[40] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," 2018.

[41] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern, T. Köppe, K. Millikin, S. Gaffney, S. Elster, J. Broshear, C. Gamble, K. Milan, R. Tung, M. Hwang, T. Cemgil, M. Barekatain, Y. Li, A. Mandhane, T. Hubert, J. Schrittwieser, D. Hassabis, P. Kohli, M. Riedmiller, O. Vinyals, and D. Silver, "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, 2023. [Online]. Available: https://doi.org/10.1038/s41586-023-06004-9

[42] C. Mendis, C. Yang, Y. Pu, S. Amarasinghe, and M. Carbin, *Compiler Auto-Vectorization with Imitation Learning.* Red Hook, NY, USA: Curran Associates Inc., 2019.

[43] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "Neurovectorizer: End-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY,

USA: Association for Computing Machinery, 2020, p. 242–255. [Online]. Available: https://doi.org/10.1145/3368826.3377928