Reducing Dynamic Checks in Auto-Vectorization using Symbolic Execution

Student Name: Manas
Roll Number: 2018244

BTP report submitted in partial fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Applied Mathematics

on 25 July 2022

BTP Track: Research

BTP Advisor
Dr. Piyus Kedia

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

I hereby declare that the work presented in the report entitled **"Reducing Dynamic Checks in Auto-Vectorization using Symbolic Execution"** submitted by me for the partial fulfillment of the requirements for the degree of *Bachelor of Technology* in *Computer Science & Applied Mathematics* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of my work carried out under guidance of **Dr. Piyus Kedia**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

.... **Place & Date: ....**

Manas

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

.... **Place & Date: ....**

Dr. Piyus Kedia

Abstract

The techniques of vectorization enable us to translate scalar code into vector form. When a collection of values of same type are being operated on by an instruction, then it is desirable to cumulate bunch of values and use a vector operation over that set. LLVM , a modular compiler infrastructure, automates this optimization over many constructs, one of which is loop. This heuristic is based upon a cost model and emission of runtime checks for establishing non-aliasing of vectors. We demonstrate the capture of non-aliasing assertions using a symbolic execution engine, KLEE. For this purpose, we have used the PolyBench/C suite.

Keywords: Auto-Vectorization, Symbolic execution, Software testing

Acknowledgments

I thank Dr. Piyus Kedia for his support and inputs.

Contents

Chapter 1

Introduction

The need for vectorization is prominent for performance enhancement, but doing vectorization by-hand is a costly and error-prone task. Optimizing compilers have invented heuristics for doing this for us. Thus, the term auto-vectorization is coined. There is no one-to-one mapping between scalar code and its vector form. Take a simple example of a for loop with an iteration space of n. It is an architectural hurdle to identify the correct induction variable to be used for the iteration space in the vectorized code.

```
for (int i = 0 ; i < n; ++i) {
    A[i] + B[i]
}
```

```
for (int i = 0 ; i < n / 4; i = i + 4) {
    <4 x int>(A[i..i + 4] + B[i..i + 4])
}
```

The optimal value for induction variable is machine-dependent.

LLVM [2] has an optimization technique called the Loop Vectorizer. This technique is based on evaluating cost model for vectorizing looping constructs. So, whenever the optimizer detects a loop, and the loop meets certain conditions for enabling vectorization, it transforms that loop into a vectorized form. But in practice, it is easier said than done. The optimizer generally encounters codebase in which there is no trivial way to ensure that the values involved in operation are disjoint. Consider the following example of three arrays with length n.

```
int *A, *B, *C;

each_malloc(sizeof(int) * n);

for (int i = 0; i < n; ++i) {
  C[i] = A[i] + B[i]
}
```

Assuming that all three pointers to integers are allocated a memory space of $sizeof(int) \times n$, it is difficult to ensure that A and B do not overlap at all when the allocation is done at runtime. Loop Vectorizer inserts, what is known as a runtime or dynamic check at the starting of loop. It checks the bounds of allocated spaces and asserts that they don't overlap. Failing to assert this, the code is run as scalar at runtime.

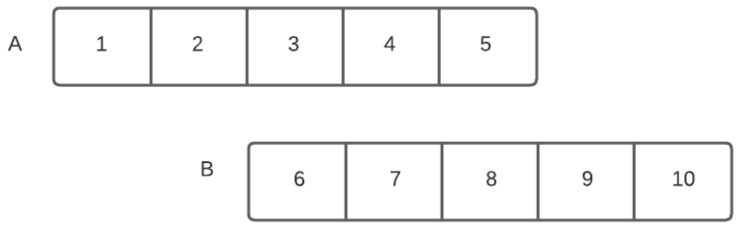


Figure 1.1: Allocated memory may overlap

These dynamic checks hinder the performance as the machine eventually needs to check the aliasing before each iteration of computation can be executed. This is costly for the current model of Loop Vectorizer. We wish to identify cases where we can ensure that the values associated are disjoint, so that we can reduce the number of dynamic checks, and thus improving performance of the vectorized code.

For this work, we rely on symbolic execution to pre-determine the judgements of non-aliasing. Symbolic execution statically traverses through the code by symbolizing variables. We have utilized KLEE symbolic execution engine for this task. KLEE operates on LLVM bitcode files, and explores all paths in the source code. KLEE assumes concrete values for the symbols defined in the source code, and generates tests cases whenever the program execution is halted.

We put assertions before the loops which ensure the disjoint-ness of associated variables, and check if KLEE can prove these assertions or not for certain test cases.

Chapter 2

Auto-Vectorization in LLVM

LLVM has implemented two vectorizers: Loop Vectorizer and SLP Vectorizer. Former operates on looping constructs, while latter merges independent instructions into vector form. Our focus is on the Loop Vectorizer. It is enabled by default in Clang. And one can disable it by using `-fno-vectorize` flag. This paper on auto-vectorization [4] by the same members who are currently working on LLVM's auto-vectorization, contains more information about the implementation.

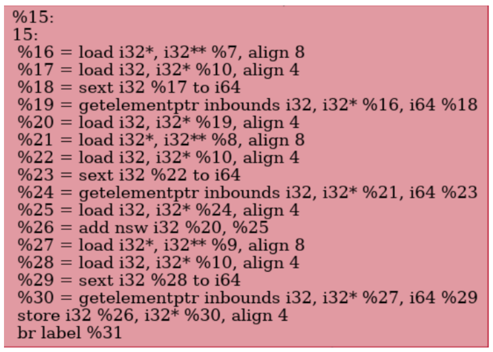
2.1 Runtime Checks

As we have established before that the Loop Vectorizer has no way of identifying whether associated variables will overlap or not at runtime, so it emits runtime checks to figure that out dynamically.

```
int *foo(int n, int *A, int *B, int *C) {
  for (int i=0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

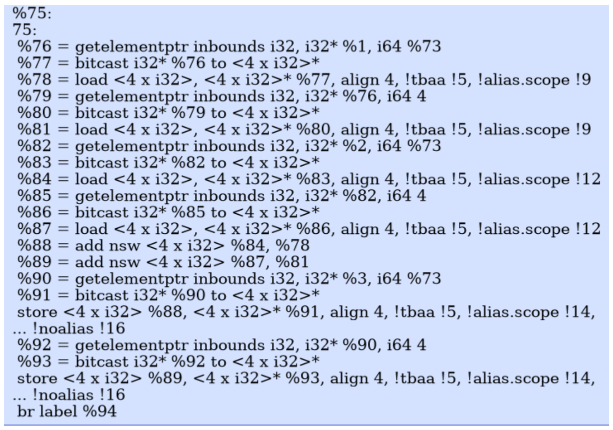
Programmatically, `restrict` keyword can be used to tell the compiler if two objects are not going to alias.

Below are the two snipped control-flow graphs of above code. First is the scalar form and second is the vectorized form. Note how convoluted the vectorized control-flow becomes even for a single loop.



```
%15:
15:
 %16 = load i32*, i32** %7, align 8
 %17 = load i32, i32* %10, align 4
 %18 = sext i32 %17 to i64
 %19 = getelementptr inbounds i32, i32* %16, i64 %18
 %20 = load i32, i32* %19, align 4
 %21 = load i32*, i32** %8, align 8
 %22 = load i32, i32* %10, align 4
 %23 = sext i32 %22 to i64
 %24 = getelementptr inbounds i32, i32* %21, i64 %23
 %25 = load i32, i32* %24, align 4
 %26 = add nsw i32 %20, %25
 %27 = load i32*, i32** %9, align 8
 %28 = load i32, i32* %10, align 4
 %29 = sext i32 %28 to i64
 %30 = getelementptr inbounds i32, i32* %27, i64 %29
 store i32 %26, i32* %30, align 4
 br label %31
```

Figure 2.1: Scalar computation



```
%75:
75:
 %76 = getelementptr inbounds i32, i32* %1, i64 %73
 %77 = bitcast i32* %76 to <4 x i32>*
 %78 = load <4 x i32>, <4 x i32>* %77, align 4, !tbaa !5, !alias.scope !9
 %79 = getelementptr inbounds i32, i32* %76, i64 4
 %80 = bitcast i32* %79 to <4 x i32>*
 %81 = load <4 x i32>, <4 x i32>* %80, align 4, !tbaa !5, !alias.scope !9
 %82 = getelementptr inbounds i32, i32* %2, i64 %73
 %83 = bitcast i32* %82 to <4 x i32>*
 %84 = load <4 x i32>, <4 x i32>* %83, align 4, !tbaa !5, !alias.scope !12
 %85 = getelementptr inbounds i32, i32* %82, i64 4
 %86 = bitcast i32* %85 to <4 x i32>*
 %87 = load <4 x i32>, <4 x i32>* %86, align 4, !tbaa !5, !alias.scope !12
 %88 = add nsw <4 x i32> %84, %78
 %89 = add nsw <4 x i32> %87, %81
 %90 = getelementptr inbounds i32, i32* %3, i64 %73
 %91 = bitcast i32* %90 to <4 x i32>*
 store <4 x i32> %88, <4 x i32>* %91, align 4, !tbaa !5, !alias.scope !14,
... !noalias !16
 %92 = getelementptr inbounds i32, i32* %90, i64 4
 %93 = bitcast i32* %92 to <4 x i32>*
 store <4 x i32> %89, <4 x i32>* %93, align 4, !tbaa !5, !alias.scope !14,
... !noalias !16
 br label %94
```

Figure 2.2: Vector computation

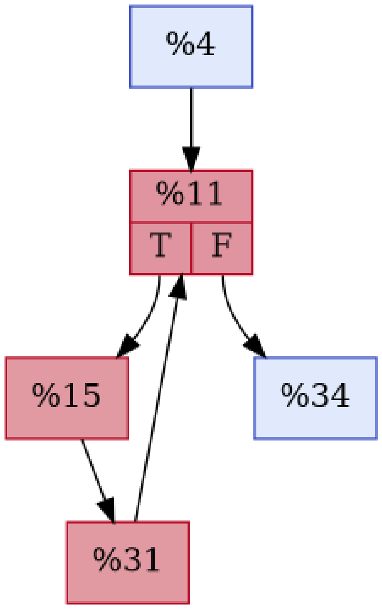


Figure 2.3: Scalar CFG

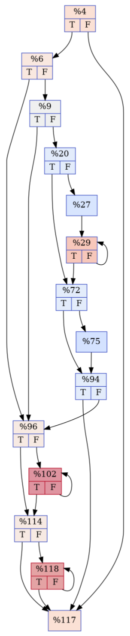


Figure 2.4: Vector CFG

2.2 Cost Model

The cost model involved in loop vectorizing process is known as LoopVectorizationCostModel. It essentially tries to predict expected speedup/slowdowns according to the supported instruction set. Using information from this method, the vectorizer finalizes whether it is optimal to transform the code or not.

Chapter 3

Symbolic Execution

This method of analyzing source code determines which inputs are causing for the execution of a path. This can be seen as a contrast to fuzzing techniques, in which code structure or control-flow is not taken into account, and program is introduced with almost random inputs. Symbolic execution, on the other hand, considers the control-flow and while exploring those paths, assign values to variables which would trigger those path executions. It helps in being more methodological while generating test cases for a program.

3.1 KLEE

KLEE [1] is a symbolic execution engine, which explore paths by generating appropriate tests and statically determines possible bugs in the program.

We demonstrate its usage on a C program (before and after vectorization).

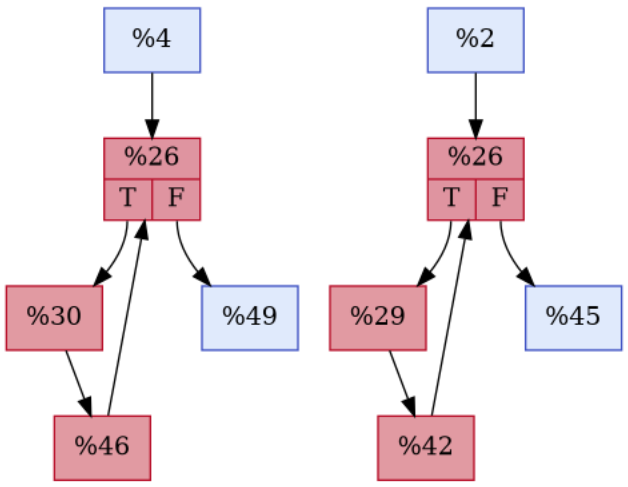


Figure 3.1: Scalar CFG for foo and main

```c
int *foo(int n, int *A, int *B, int *C) {
  // Putting invariants to ensure arrays
  // don't overlap
  klee_assume(A + n < B);
  klee_assume(B + n < C);

  for (int i=0; i < n; ++i) {
    C[i] = A[i] + B[i];

    // KLEE will halt execution on Assert
    if (A + n < B && B + n < C) {
      assert(0);
    }
  }
  return C;
}
```

```c
int main(int argc, char **argv) {
  int n = 4;

  int *A = malloc(sizeof(A) * n);
  int *B = malloc(sizeof(B) * n);
  int *C = malloc(sizeof(C) * n);

  for (int i=0; i < 4 ; ++i) {
    A[i] = i + 1;
    B[i] = 10 - i;
  }

  klee_make_symbolic(&A, sizeof(A), "A");
  klee_make_symbolic(&B, sizeof(B), "B");
  klee_make_symbolic(&C, sizeof(C), "C");

  return *foo(n, A, B, C);
}
```

This program symbolizes all three arrays, and `klee_assume` calls ensure that those symbols, when given values, do not overlap with each other. The assertion inside the for loop checks if arrays are disjoint at runtime as well. By running KLEE over this program, we can find those test cases where KLEE is able to hit the assertion. These tests point to us the cases where KLEE is unable to ensure non-aliasing symbolically.

Over the scalar code, KLEE exited with the following information:

```
Elapsed: 00:00:59
KLEE: done: explored paths = 166
KLEE: done: avg. constructs per query = 139
KLEE: done: total queries = 1923
KLEE: done: valid queries = 411
KLEE: done: invalid queries = 1512
KLEE: done: query cex = 1923

KLEE: done: total instructions = 385
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 166
KLEE: done: generated tests = 5
```

And on the vectorized counterpart, it performed as following:

```
Elapsed: 00:01:21
KLEE: done: explored paths = 181
KLEE: done: avg. constructs per query = 141
KLEE: done: total queries = 2430
KLEE: done: valid queries = 570
KLEE: done: invalid queries = 1860
KLEE: done: query cex = 2430

KLEE: done: total instructions = 415
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 181
KLEE: done: generated tests = 6
```

These results show that KLEE was able to hit the assertions, hence, was able to find test cases where these assertions will hold.

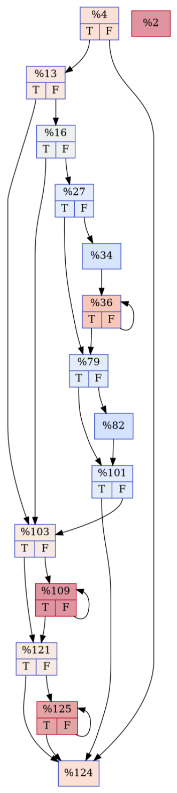


Figure 3.2: Vectorized CFG for foo and main

Chapter 4

Benchmark & Observations

4.1 PolyBench/C

It [5] is a C version of benchmark suite consisting of 30 numerical computations from the domains of linear algebra, image processing, physics simulation, dynamic programming, statistics, and few others. These benchmark tries to simulate the real world programming problems which generally use looping construcuts. This fits our need of a suite on which we can apply auto-vectorization. Apart from the standard files, the following changes have been made to ensure the functioning of KLEE over this suite [3].

1. A bash script `utilities/run-klee.sh` is provided which generates LLVM bitcode files for each C source in the benchmark. These bitcodes are then analyzed by KLEE one by one.
2. Each source in the benchmark is modified to introduce `klee_make_symbolic` and `klee_assume` calls. As we know, the first function symbolizes associated parameter, while second function ensures certain assertions to be held by KLEE during its symbolic execution.

4.2 KLEE on PolyBench/C

Modifying all C files as mentioned above to enable symbolizing arrays, we build the suite with `clang-13`. The following special parameters, such as, `-Xclang -disable-O0-optnone` to enable optimizations, were used. During the invocation of KLEE, optimizations were disabled, because we previously had optimized via clang itself. All external calls were enabled to allow various `printf` and other functions to be treated by KLEE as warnings and not errors. To reduce the number of explored states, `--posix-runtime` was disabled, and

`--only-output-states-covering-new` was enabled. Disabling posix runtime ensured that KLEE will not explore paths in posix libraries, and it will only dump tests for newly explored paths.

Chapter 5

Conclusion

This demonstratation concludes that out of 30 C source files in the PolyBench/C suite, KLEE was able to generate tests for 8 programs, which were able to assert the non-aliasing of arrays. Apart from these, one program `./stencils/seidel-2d/seidel-2d.c` was exited on invalid free pointer error. The rest were exited due to `out-of-bounds` error.

The list of files which asserted non-aliasing were:

- ./linear-algebra/kernels/2mm/2mm.c
- ./linear-algebra/kernels/3mm/3mm.c
- ./linear-algebra/kernels/atax/atax.c
- ./linear-algebra/kernels/bicg/bicg.c
- ./linear-algebra/kernels/doitgen/doitgen.c
- ./linear-algebra/kernels/mvt/mvt.c
- ./linear-algebra/blas/gemm/gemm.c
- ./linear-algebra/blas/gemver/gemver.c

Bibliography

[1] Cadar, C., Dunbar, D., Engler, D. R., et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[2] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[3] Louis-Noël Pouchet, M. PolyBench/C. `https://github.com/weirdsmiley/polybench-c-4.2.1-beta`, 2022.

[4] Nuzman, D., Rosen, I., and Zaks, A. Auto-vectorization of interleaved data for simd. *ACM SIGPLAN Notices 41*, 6 (2006), 132–143.

[5] Pouchet, L.-N. PolyBench/C. `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`, 2015.